目錄

Introduction 1.1
第一部分 初见shell 1.2
1. 为什么使用shell编程 1.2.1
2. 和Sha-Bang(#!)一起出发 1.2.2
2.1 调用一个脚本 1.2.2.1
2.2 牛刀小试 1.2.2.2
第二部分 shell基础 1.3
3. 特殊字符 1.3.1
4. 变量与参数 1.3.2
4.1 变量替换 1.3.2.1
4.2 变量赋值 1.3.2.2
4.3 Bash弱类型变量 1.3.2.3
4.4 特殊变量类型 1.3.2.4
5. 引用 1.3.3
5.1 引用变量 1.3.3.1
5.2 转义 1.3.3.2
6. 退出与退出状态 1.3.4
7. 测试 1.3.5
7.1 测试结构 1.3.5.1
7.2 文件测试操作 1.3.5.2
7.3 其他比较操作 1.3.5.3
7.4 嵌套 if/then 条件测试 1.3.5.4
7.5 牛刀小试 1.3.5.5
8. 运算符相关话题 1.3.6
8.1 运算符 1.3.6.1
8.2 数字常量 1.3.6.2
8.3 双圆括号结构 1.3.6.3

8.4 运算符优先级 1.3.6.4
第三部分 shell进阶 1.4
9. 换个角度看变量 1.4.1
9.1 内部变量 1.4.1.1
9.2 指定变量属性： 或 1.4.1.2
9.3 ：随机产生整数 1.4.1.3
10. 变量处理 1.4.2
10.1 字符串处理 1.4.2.1
10.1.1 使用 awk 处理字符串 1.4.2.1.1
10.1.2 参考资料 1.4.2.1.2
10.2 参数替换 1.4.2.2
11. 循环与分支 1.4.3
11.1 循环 1.4.3.1
11.2 嵌套循环 1.4.3.2
11.3 循环控制 1.4.3.3
11.4 测试与分支 1.4.3.4
12. 命令替换 1.4.4
13. 算术扩展 1.4.5
14. 休息时间 1.4.6
第五部分 进阶话题 1.5
19. 嵌入文档 1.5.1
20. I/O 重定向 1.5.2
20.1 使用 exec 1.5.2.1
20.2 重定向代码块 1.5.2.2
20.3 应用程序 1.5.2.3
22. 限制模式的Shell 1.5.2.4
23. 进程替换 1.5.2.5
26. 列表结构 1.5.2.6
25. 别名 1.5.3

《Advanced Bash-Scripting Guide》 in Chinese

《高级Bash脚本编程指南》Revision 10中文版

联系/加入我们

- 邮箱：absguide#linuxstory.org(将#替换为@)
- QQ群：535442421

原著及早期翻译作品

原著

- 原著链接：http://tldp.org/LDP/abs/html/
- 原作：Mendel Cooper
- 原著版本：Revision 10, 10 Mar 2014

译著

- 早期译著连接：http://www.linuxsir.org/bbs/thread256887.html
- 译者：杨春敏 黄毅
- 译著版本：Revision 3.7, 23 Oct 2005
- 最新 Revision 10 由 Linux Story 社区的 imcmy 同学发起并组织翻译
- Linux Story 通告地址 ：http://www.linuxstory.org/asdvanced-bash-scripting-guide-in-chinese/

翻译作品

翻译作品放在GitBook上，欢迎阅读！

翻译进度

- 第一部分 初见Shell[@imcmy][@zihengcat]
    - 1. 为什么使用shell编程[@imcmy][@zihengcat]
    - 2. Sha-Bang（#!）一起出发[@imcmy][@zihengcat]
- 第二部分 Shell基础[@imcmy][@zihengcat]
    - 3. 特殊字符[@imcmy][@zihengcat]
    - 4. 变量与参数[@imcmy][@zihengcat]
    - 5. 引用[@mr253727942][@zihengcat]
    - 6. 退出与退出状态[@samita2030][@zihengcat]
    - 7. 测试[@imcmy][@zihengcat]
    - 8. 运算符相关话题[@samita2030][@zihengcat]
- 第三部分 Shell进阶[@imcmy]
    - 9. Another Look at Variables[@Ninestd]
    - 10. 变量处理[@imcmy]
    - 11. 循环与分支[@imcmy]
    - 12. 命令替换[@imcmy]
    - 13. 算术扩展[@imcmy]
    - 14. 休息时间[@imcmy]
- 第四部分. 命令[@zhaozq]
    - 15. 内建命令[@zhaozq]
    - 16. 外部过滤器，程序与命令[@zhaozq]
    - 17. 系统与高级命令[@zhaozq]
- 第五章. Advanced Topics
    - 18. 正则表达式[@Zjie]
        - 18.1 正则表达式简介[@Zjie]
        - 18.2 文件名替换[@Zjie]
    - 19. 嵌入文档[@mingmings]
    - 20. I/O 重定向[@mingmings]
    - 21. Subshells[@mingmings]
    - 22. Restricted Shells[@panblack]
    - 23. Process Substitution[@panblack]
    - 24. Functions[@zy416548283]
    - 25. 别名[@mingmings]
    - 26. List Constructs[@panblack]
    - 27. Arrays[@zy416548283]
    - 28. Indirect References[@panblack]
    - 29. /dev and /proc[@panblack]

  - 30. Network Programming[@Zjie]
  - 31. Of Zeros and Nulls[@panblack]
  - 32. Debugging[@wuqichao]
  - 33. Options[@zy416548283]
  - 34. Gotchas[@liuburn]
  - 35. Scripting With Style[@chuchingkai]
  - 36. Miscellany[@richard-ma]
  - 37. Bash, versions 2, 3, and 4
  - 38. Endnotes[@zy416548283]
    - 38.1 Author's Note
    - 38.2 About the Author
    - 38.3 Where to Go For Help
    - 38.4 Tools Used to Produce This Book
    - 38.5 Credits
    - 38.6 Disclaimer
- Bibliography
- Appendix
  - A. Contributed Scripts
  - B. Reference Cards
  - C. A Sed and Awk Micro-Primer[@wuqichao]
    - C.1 Sed[@wuqichao]
    - C.2 Awk[@wuqichao]
  - D. Parsing and Managing Pathnames
  - E. Exit Codes With Special Meanings
  - F. A Detailed Introduction to I/O and I/O Redirection
  - G. Command-Line Options
    - G.1 Standard Command-Line Options
    - G.2 Bash Command-Line Options
  - H. Important Files
  - I. Important System Directories
  - J. An Introduction to Programmable Completion
  - K. Localization
  - L. History Commands
  - M. Sample .bashrc and .bash_profile Files
  - N. Converting DOS Batch Files to Shell Scripts
  - O. Exercises

        - O.1 Analyzing Scripts
        - O.2 Writing Scripts
    - P. Revision History
    - Q. Download and Mirror Sites
    - R. To Do List
    - S. Copyright
    - T. ASCII Table
- Index
- List of Tables
- List of Examples

翻译校审流程

初始化

1. 首先fork项目
2. 把fork过去的项目clone到本地
3. 命令行下运行 `git checkout -b dev` 创建一个新分支
4. 运行 `git remote add upstream https://github.com/LinuxStory/Advanced-Bash-Scripting-Guide-in-Chinese.git` 添加远端库
5. 运行 `git remote update` 更新
6. 运行 `git fetch upstream master` 拉取更新到本地
7. 运行 `git rebase upstream/master` 将更新合并到你的分支

初始化只需要做一遍，之后请在dev分支进行修改。

如果修改过程中项目有更新，请重复5、6、7步。

翻译校审流程

1. 保证在dev分支中
2. 打开README.md，在翻译进度后加上你自己的github名
    > 1. Shell Programming! [@翻译人][@校审人]
3. 本地提交修改，写明提交信息
4. push到你fork的项目中，然后登录GitHub

5. 在你fork的项目的首页可以看到一个 `pull request` 按钮，点击它，填写说明信息，然后提交即可

   > 为了不重复工作，请等待我们确认了你的pull request(即你的名字出现在项目中时)，再进行翻译校审工作

6. 进行翻译校审，重复3-5步提交翻译校审的作品

> 新手可以参阅针对github小白的《翻译流程详解》,妹子写的呦～

翻译校审建议

1. 使用markdown进行翻译校审，文件名必须使用英文
2. 翻译校审后的文档请放到source文件夹下的对应章节中，然后pull request即可
3. 有任何问题随时欢迎发issue
4. 术语尽量保证和已翻译的一致，也可以查询微软术语搜索或Linux中国术语词典
5. 你可以将你认为是术语的词汇加入术语表 `TERM.md` 中

关于版权

根据原著作者的要求，翻译成果属于公有领域(CC0)，翻译参与人员及原著作者Mendel Cooper享有署名权

第一部分 初见Shell

> 脚本：文章；书面文档
>
> ——韦伯斯特字典*1913*年版

Shell是一种命令解释器，它不仅分离了用户层与操作系统内核，更是一门强大的编程语言。我们称为shell编写的程序为脚本（script）。脚本是一种易于使用的工具，它能够将系统调用、工具软件、实用程序（utility）和已编译的二进制文件联系在一起构建程序。实际上，shell脚本可以调用所有的UNIX命令、实用程序以及工具软件。如果你觉得这还不够，使用像 `test` 命令和循环结构这样的shell内建命令能够让脚本更加灵活强大。Shell脚本特别适合完成系统管理任务和那些不需要复杂结构性语言实现的重复工作。

内容目录

- 1. 为什么使用shell编程
- 2. 和Sha-Bang（#!）一起出发
  - 2.1 调用一个脚本
  - 2.2 牛刀小试

第一章 为什么使用shell编程

> 没有任何一种程序设计语言是完美的，甚至没有一个最好的语言。只有在特定环境下适合的语言。
>
> —— Herbert Mayer

无论你是否打算真正编写shell脚本，只要你想要在一定程度上熟悉系统管理，了解掌握shell脚本的相关知识都是非常有必要的。例如Linux系统在启动的时候会执行 `/etc/rc.d` 目录下的shell脚本来恢复系统配置和准备服务。详细了解这些启动脚本对分析系统行为大有益处，何况，你很有可能会去修改它们呢。

编写shell脚本并不困难，shell脚本由许多小的部分组成，而其中只有数量相当少的与shell本身特性，操作和选项[1]有关的部分才需要去学习。Shell语法非常简单朴素，很像是在命令行中调用和连接工具，你只需遵循很少一部分的"规则"就可以了。大部分短小的脚本通常在第一次就可以正常工作，即使是一个稍长一些的脚本，调试起来也十分简单。

> 在个人计算机发展的早期，BASIC语言让计算机专业人士能够在早期的微机上编写程序。几十年后，Bash脚本可以让所有仅对Linux或UNIX系统有初步了解的用户在现代计算机上做同样的事。
>
> 我们现在已经可以做出一些又小又快的单板机，比如树莓派。Bash脚本提供了一种发掘这些有趣设备潜力的方式。

使用shell脚本构建一个复杂应用原型（prototype），不失为是一种虽有缺陷但非常快速的方式。在项目开发初期，使用脚本实现部分功能往往显得十分有用。在使用C/C++，Java，Perl或Python编写最终代码前，可以使用shell脚本测试，修补应用结构，提前发现重大缺陷。

Shell脚本与经典的UINX哲学相似，将复杂的任务划分为简单的子任务，将组件与工具连接起来。许多人认为比起新一代功能强大、高度集成的语言，例如Perl，shell脚本至少是一种在美学上更加令人愉悦的解决问题的方式，Perl试图做到面面俱到，但你必须强迫自己改变思维方式适应它。

Herbert Mayer曾说：“有用的语言需要数组、指针以及构建数据结构的通用机制”。如果依据这些标准，那shell脚本距“有用”还差得很远，甚至是“无用”的。

什么时候不应该使用shell脚本

- 资源密集型的任务，尤其是对速度有要求（如排序、散列、递归[2]等）
- 需要做大量的数学运算，例如浮点数运算，高精度运算或者复数运算（使用C++或FORTRAN代替）
- 有跨平台需求（使用C或者Java代替）
- 必须使用结构化编程的复杂应用（如变量类型检查、函数原型等）
- 影响系统全局的关键性任务
- 对安全性有高要求，需要保证系统的完整性以及阻止入侵、破解、恶意破坏
- 项目包含有连锁依赖关系的组件
- 需要大量的文件操作（Bash只能访问连续的文件，并且是以一种非常笨拙且低效的逐行访问的方式进行的）
- 需要使用多维数组
- 需要使用如链表、树等数据结构
- 需要产生或操作图像和图形用户接口（GUI）
- 需要直接访问系统硬件或外部设备
- 需要使用端口或套接字输入输出端口（Socket I/O）
- 需要使用库或旧程序的接口
- 私有或闭源的项目（Shell脚本直接将源代码公开，所有人都可以看到）

如果你的应用满足上述任意一条，你可以考虑使用更加强大的脚本语言，如Perl，Tcl，Python，Ruby等，或考虑使用编译型语言，如C，C++或Java等。即使如此，在开发阶段使用shell脚本建立应用原型也是十分有用的。

我们接下来将使用Bash。Bash是"Bourne-Again shell"的首字母缩略词[3]，Bash来源于Stephen Bourne开发的Bourne shell（sh）。如今Bash已成为了大部分UNIX衍生版中shell脚本事实上的标准。本书所涉及的大部分概念在其他shell中也是适用的，例如Korn Shell，Bash从它当中继承了一部分的特性[4]；又如C Shell及其变体（需要注意的是，1993年10月Tom Christiansen在Usenet帖子中指出，因C Shell内部固有的问题，不推荐使用C Shell编程）

接下来的部分将是一些编写shell脚本的指导。这些指导很大程度上依赖于实例来阐述shell的特性。本书所有的例子都能够正常工作，并在尽可能的范围内进行过测试，其中的一部分已经运用在实际生产生活中。读者们可以使用这些在存档中的例子（文件名为 `scriptname.sh` 或 `scriptname.bash` ）[5]，赋予它们可执行权限（ `chmod u+rx scriptname` ），然后执行它们看看会发生什么。如果存档不可

用，读者朋友也可以从本书的HTML或者PDF版本中复制粘贴代码出来。需要注意的是，在部分例子中使用了一些暂时还未被解释的特性，这需要读者暂时跳过它们。

除特别说明，本书所有例子均由本书作者编写。

> His countenance was bold and bashed not.
>
> —— Edmund Spenser

> 1. 这些操作和选项被称为内建命令（builtin），是shell的内部特征。↩

> 2. 虽然递归可以在shell脚本中实现，但是它的效率很低且实现起来很复杂、不具有美感。↩

> 3. 首字母缩略词是由每一个单词的首字母拼接而成的易读的代替短语。这不是一个好习惯，通常会引起一些不必要的麻烦。↩

> 4. ksh88中的许多特性，甚至一些ksh93的特性都被合并到Bash中了。↩

> 5. 按照惯例，用户编写的Bourne shell脚本应该在文件名后加上 `.sh` 的扩展名。而那些系统脚本，比如在 `/etc/rc.d` 中的脚本通常不遵循这种规范。↩

第二章 和Sha-Bang（#!）一起出发

> Shell编程声名显赫
>
> —— Larry Wall

本章目录

- 2.1 调用一个脚本
- 2.2 牛刀小试

一个最简单的脚本其实就是将一连串系统命令存储在一个文件中。最起码，它能帮你省下重复输入这一连串命令的功夫。

样例 2-1. *cleanup*：清理 `/var/log` 目录下的日志文件

```
# Cleanup
# 请使用root权限执行

cd /var/log
cat /dev/null > messages
cat /dev/null > wtmp
echo "Log files cleaned up."
```

这支脚本仅仅是一些可以很容易从终端或控制台输入的命令的集合罢了，没什么特殊的地方。将命令放在脚本中的好处是，你不用再一遍遍重复输入这些命令啦。脚本成了一支程序、一款工具，它可以很容易的被修改或为特殊需求定制。

样例 2-2. *cleanup*：改进的清理脚本

```
#!/bin/bash
# Bash脚本标准起始行。

# Cleanup, version 2

# 请使用root权限执行。
# 这里可以插入代码来打印错误信息，并在未使用root权限时退出。

LOG_DIR=/var/log
# 使用变量比硬编码（hard-coded）更合适
cd $LOG_DIR

cat /dev/null > messages
cat /dev/null > wtmp


echo "Logs cleaned up."

exit # 正确终止脚本的方式。
     # 不带参数的exit返回上一条指令的运行结果。
```

现在我们看到了一个真正意义上的脚本! 让我们继续前进...

样例 2-3. *cleanup*：改良、通用版

```
#!/bin/bash
# Cleanup, version 3

# 注意：
# --------
# 此脚本涉及到许多后边才会解释的特性。
# 当你阅读完整本书的一半以后，理解它们就没有任何困难了。



LOG_DIR=/var/log
ROOT_UID=0       # UID为0的用户才拥有root权限。
LINES=50         # 默认保存messages日志文件行数。
E_XCD=86         # 无法切换工作目录的错误码。
E_NOTROOT=87     # 非root权限用户执行的错误码。
```

```
# 请使用root权限运行。
if [ "$UID" -ne "$ROOT_UID" ]
then
  echo "Must be root to run this script."
  exit $E_NOTROOT
fi

if [ -n "$1" ]
# 测试命令行参数（保存行数）是否为空
then
  lines=$1
else
  lines=$LINES # 如果为空则使用默认设置
fi


#  Stephane Chazelas 建议使用如下方法检查命令行参数，
#  但是这已经超出了此阶段教程的范围。
#
#    E_WRONGARGS=85  # Non-numerical argument (bad argument form
at).
#    case "$1" in
#    ""      ) lines=50;;
#    *[!0-9]*) echo "Usage: `basename $0` lines-to-cleanup";
#     exit $E_WRONGARGS;;
#    *       ) lines=$1;;
#    esac
#
#* 在第十一章“循环与分支”中会对此作详细的阐述。


cd $LOG_DIR

if [ `pwd` != "$LOG_DIR" ]  # 也可以这样写 if [ "$PWD" != "$LOG_DI
R" ]
                            # 检查工作目录是否为 /var/log ?
then
```

```
  echo "Can't change to $LOG_DIR"
  exit $E_XCD
fi  # 在清理日志前，二次确认是否在正确的工作目录下。

# 更高效的写法：
#
# cd /var/log || {
#   echo "Cannot change to necessary directory." >&2
#   exit $E_XCD;
# }


tail -n $lines messages > mesg.temp # 保存messages日志文件最后一部分
mv mesg.temp messages              # 替换系统日志文件以达到清理目的

#  cat /dev/null > messages
#* 我们不需要使用这个方法了，上面的方法更安全

cat /dev/null > wtmp  #  ': > wtmp' 与 '> wtmp' 有同样的效果
echo "Log files cleaned up."
#  注意在/var/log目录下的其他日志文件不会被这个脚本清除

exit 0
#  返回0表示脚本运行成功
```

也许你并不希望清空全部的系统日志，这个脚本保留了messages日志的最后一部分。随着学习的深入，你将明白更多提高脚本运行效率的方法。

脚本起始行*sha-bang*（#!）[1]告诉系统这个脚本文件需要使用指定的命令解释器来执行。#!实际上是一个占两字节[2]的幻数（magic number），幻数可以用来标识特殊的文件类型，在这里则是标记可执行shell脚本（你可以在终端中输入 `man magic` 了解更多信息）。紧随#!的是一个路径名。此路径指向用来解释此脚本的程序，它可以是shell，可以是程序设计语言，也可以是实用程序。这个解释器从头（#!的下一行）开始执行整个脚本的命令，同时忽略注释。[3]

```
#!/bin/sh
#!/bin/bash
#!/usr/bin/perl
#!/usr/bin/tcl
#!/bin/sed -f
#!/bin/awk -f
```

上面每一条脚本起始行都调用了不同的解释器，比如 `/bin/sh` 调用了系统默认shell（Linux系统中默认是bash）[4]。大部分UNIX商业发行版中默认的是Bourne shell，即 `#!/bin/sh` 。你可以以牺牲Bash特性为代价，在非Linux的机器上运行sh脚本。当然，脚本得遵循POSIX[5] sh标准。

需要注意的是 `#!` 后的路径必须正确，否则当你运行脚本时只会得到一条错误信息，通常是"Command not found."[6]

当脚本仅包含一些通用的系统命令而不使用shell内部指令时，可以省略 `#!` 。第三个例子需要 `#!` 是因为当对变量赋值时，例如 `lines=50` ，使用了与shell特性相关的结构[7]。再重复一次， `#!/bin/sh` 调用的是系统默认shell解释器，在Linux系统中默认为 `/bin/bash` 。

这个例子鼓励读者使用模块化的方式编写脚本，并在平时记录和收集一些在以后可能会用到的代码模板。最终你将拥有一个相当丰富易用的代码库。以下的代码可以用来测试脚本被调用时的参数数量是否正确。

```
E_WRONG_ARGS=85
script_parameters="-a -h -m -z"
                 # -a = all, -h = help 等等

if [ $# -ne $Number_of_expected_args ]
then
  echo "Usage: `basename $0` $script_parameters"
  # `basename $0` 是脚本的文件名
  exit $E_WRONG_ARGS
fi
```

大多数情况下，你会针对特定的任务编写脚本。本章的第一个脚本就是这样。然后你也许会泛化（generalize）脚本使其能够适应更多相似的任务，比如用变量代替硬编码，用函数代替重复代码。

1

> 1. 在文献中更常见的形式是she-bang或者sh-bang。它们都来源于词汇sharp(#)和bang(!)的连接。 ↩

> 2. 一些UNIX的衍生版（基于4.2 BSD）声称他们使用四字节的幻数，在#!后增加一个空格，即 `#! /bin/sh` 。而Sven Mascheck指出这是虚构的。 ↩

> 3.
> 命令解释器首先将会解释#!这一行，而因为#!以#打头，因此解释器将其视作注释。起始行作为调用解释器的作用已经完成了。
>
> 事实上即使脚本中含有不止一个#!,bash也会将除第一个`#!`以外的解释为注释。
>
> ```
> #!/bin/bash
>
> echo "Part 1 of script."
> a=1
>
> #!/bin/bash
> # 这并不会启动新的脚本
>
> echo "Part 2 of script."
> echo $a  # $a的值仍旧为1
> ```
>
> ↩

4

> 4.
> 这里允许使用一些技巧。
>
> ```
> #!/bin/rm
> # 自我删除的脚本
>
> # 当你运行这个脚本，除了这个脚本本身消失以外并不会发生什么。
>
> WHATEVER=85
>
> echo "This line will never print (betcha!)."
>
> exit $WHATEVER  # 这没有任何关系。脚本将不会从这里退出。
>                 # 尝试在脚本终止后打印echo $a。
>                 # 得到的值将会是0而不是85.
> ```
>
> 当然你也可以建立一个起始行是 `#!/bin/more` 的README文件，并且使它可以执行。结果就是这个文件成为了一个可以打印本身的文件。（查看样例 19-3，使用 `cat` 命令的here document也许是一个更好的选择）↩

> 5. 可移植操作系统接口（POSIX）尝试标准化类UNIX操作系统。POSIX规范可以在Open Group site中查看。↩

> 6. 为了避免这种情况的发生，可以使用 `#!/bin/env bash` 作为起始行。这在bash不在 `/bin` 的UNIX系统中会有效果。↩

> 7. 如果bash是系统默认shell，那么脚本并不一定需要#!作为起始行。但是当你在其他的shell中运行脚本，例如tcsh，则需要使用#!。↩

2.1 调用一个脚本

写完一个脚本以后，你可以通过 `sh scriptname` [1]或 `bash scriptname` 来调用它（不推荐使用 `sh <scriptname` 调用脚本，因为这会禁用脚本从标准输入（stdin）读入数据）。更方便的方式是使用 `chmod` 命令使脚本可以被直接执行。

执行命令：

`chmod 555 scriptname` （给予所有用户读取/执行的权限）[2]

或

`chmod +rx scriptname` （给予所有用户读取/执行的权限）

`chmod u+rx scriptname` （仅给予脚本所有者读取/执行的权限）

当脚本的权限被设置好后，你就可以直接使用 `./scriptname` [3]进行调用测试了。如果脚本文件以sha-bang开头，那么它将自动调用指定的命令解释器运行脚本。

完成调试与测试后，你可能会将脚本文件移至 `/usr/local/bin` （使用root权限）中，使脚本可以被所有用户调用。这时你可以直接在命令行中输入 `scriptname [ENTER]` 执行脚本。

> 1. 注意，当你使用 `sh scriptname` 调用*Bash*脚本时，将会禁用与Bash特性相关的功能，脚本有可能会执行失败。↩

> 2. 脚本需要同时具有读取和执行的权限，因为shell需要读取脚本执行。↩

> 3. 为什么不直接使用 `scriptname` 来调用脚本？为什么当工作目录（$PWD）正好是 `scriptname` 所在目录时也不起作用？因为一些安全原因，当前目录（ `./` ）并不会被默认添加到用户的$PATH路径中。因此需要用户显式使用 `./scriptname` 在当前目录下调用脚本。↩

2.2 牛刀小试

1. 系统管理员通常会写一些脚本来完成自动化工作。试举例说明使用脚本的便利之处。
2. 请尝试写一个脚本。调用脚本，会打印当前系统时间和日期，所有已登录的用户和系统运行时间。并将这些信息保存到一个日志文件中。

第二部分 shell基础

目录

- 3. 特殊字符
- 4. 变量与参数
    - 4.1 变量替换
    - 4.2 变量赋值
    - 4.3 Bash弱类型变量
    - 4.4 特殊变量类型
- 5. 引用
    - 5.1 引用变量
    - 5.2 转义
- 6. 退出与退出状态
- 7. 测试
    - 7.1 测试结构
    - 7.2 文件测试操作
    - 7.3 其他比较操作
    - 7.4 嵌套 if/then 条件测试
    - 7.5 牛刀小试
- 8. 运算符和相关话题
    - 8.1 运算符
    - 8.2 数字常量
    - 8.3 双圆括号结构
    - 8.4 运算符优先级

第三章 特殊字符

是什么让一个字符变得特殊呢？如果一个字符不仅具有字面意义，而且具有元意（*meta-meaning*），我们就称它为特殊字符。特殊字符同命令和关键词（keywords）一样，是bash脚本的组成部分。

你在脚本或其他地方都能够找到特殊字符。

#

注释符。如果一行脚本的开头是#（除了#!），那么代表这一行是注释，不会被执行。

```
# 这是一行注释
```

注释也可能会在一行命令结束之后出现。

```
echo "A comment will follow." # 这儿可以写注释
#                            ^ 注意在#之前有空格
```

注释也可以出现在一行开头的空白符（whitespace）之后。

```
    # 这个注释前面存在着一个制表符（tab）
```

注释甚至可以嵌入到管道命令（pipe）之中。

```
initial=( `cat "$startfile" | sed -e '/#/d' | tr -d '\n' |\
# 删除所有带'#'注释符号的行
           sed -e 's/\./\. /g' -e 's/_/_ /g'` )
# 摘录自脚本 life.sh
```

> 命令不能写在同一行注释之后。因为没有任何方法可以结束注释(仅支持单行注释)，为了让新命令正常执行，另起一行写吧。
>
> 当然，在 `echo` 语句中被引用或被转义的#不会被认为是注释。同样，在某些参数替换式或常量表达式中的#也不会被认为是注释。

```
echo "The # here does not begin a comment."
echo 'The # here does not begin a comment.'
echo The \# here does not begin a comment.
echo The # here begins a comment.

echo ${PATH#*:}       # 参数替换而非注释
echo $(( 2#101011 ))  # 进制转换而非注释

# 感谢S.C.
```

因为引用符和转义符（"'\）转义了#。

一些模式匹配操作同样使用了#。

;

命令分隔符[分号]。允许在同一行内放置两条或更多的命令。

```
echo hello; echo there

if [ -x "$filename" ]; then    #  注意在分号以后有一个空格
#+                   ^^
  echo "File $filename exists."; cp $filename $filename.bak
else   #                       ^^
  echo "File $filename not found."; touch $filename
fi; echo "File test complete."
```

注意有时候";"需要被转义才能正常工作。

;;

`case` 条件语句终止符[双分号]。

```
case "$variable" in
  abc)  echo "\$variable = abc" ;;
  xyz)  echo "\$variable = xyz" ;;
esac
```

;;&, ;&

`case` 条件语句终止符（Bash4+ 版本）。

.

句点命令[句点]。等价于 `source` 命令（查看样例 15-22）。这是一个bash的内建命令。

.

句点可以作为文件名的一部分。如果它在文件名开头，那说明此文件是隐藏文件。使用不带参数的 `ls` 命令不会显示隐藏文件。

```
bash$ touch .hidden-file
bash$ ls -l
total 10
 -rw-r--r--    1 bozo      4034 Jul 18 22:04 data1.addressbook
 -rw-r--r--    1 bozo      4602 May 25 13:58 data1.addressbook.b
ak
 -rw-r--r--    1 bozo       877 Dec 17  2000 employment.addressb
ook


bash$ ls -al
total 14
 drwxrwxr-x    2 bozo  bozo      1024 Aug 29 20:54 ./
 drwx------   52 bozo  bozo      3072 Aug 29 20:51 ../
 -rw-r--r--    1 bozo  bozo      4034 Jul 18 22:04 data1.address
book
 -rw-r--r--    1 bozo  bozo      4602 May 25 13:58 data1.address
book.bak
 -rw-r--r--    1 bozo  bozo       877 Dec 17  2000 employment.ad
dressbook
 -rw-rw-r--    1 bozo  bozo         0 Aug 29 20:54 .hidden-file
```

当句点出现在目录中时，单个句点代表当前工作目录，两个句点代表上级目录。

```
bash$ pwd
/home/bozo/projects

bash$ cd .
bash$ pwd
/home/bozo/projects

bash$ cd ..
bash$ pwd
/home/bozo/
```

句点通常代表文件移动的目的地（目录），下式代表的是当前目录。

```
bash$ cp /home/bozo/current_work/junk/* .
```

> 复制所有的“垃圾文件”到 `当前目录`

.

句点匹配符。在正则表达式中，点号意味着匹配任意单个字符。

"

部分引用[双引号]。在字符串中保留大部分特殊字符。详细内容将在第五章介绍。

'

全引用[单引号]。在字符串中保留所有的特殊字符。是部分引用的强化版。详细内容将在第五章介绍。

,

逗号运算符。逗号运算符[1]将一系列的算术表达式串联在一起。算术表达式依次被执行，但只返回最后一个表达式的值。

```
let "t2 = ((a = 9, 15 / 3))"
# a被赋值为9，t2被赋值为15 / 3
```

逗号运算符也可以用来连接字符串。

```
for file in /{,usr/}bin/*calc
#             ^    在 /bin 与 /usr/bin 目录中
#+                 找到所有的以"calc"结尾的可执行文件
do
        if [ -x "$file" ]
        then
          echo $file
        fi
done

# /bin/ipcalc
# /usr/bin/kcalc
# /usr/bin/oidcalc
# /usr/bin/oocalc

# 感谢Rory Winston提供的执行结果
```

,, ,

在参数替换中进行小写字母转换（Bash4 新增）。

\

转义符[反斜杠]。转义某字符的标志。

`\x` 转义了字符x。双引号""内的X与单引号内的X具有同样的效果。 转义符也可以用来转义"与'，使它们表达其字面含义。

第五章将更加深入的解释转义字符。

/

文件路径分隔符[正斜杠]。起分割路径的作用。（比如 `/home/bozo/projects/Makefile` ）

它也在算术运算中充当除法运算符。

`

命令替换符。 `` `command` `` 结构可以使得命令的输出结果赋值给一个变量。通常也被称作后引号（backquotes）或反引号（backticks）。

:

空命令[冒号]。它在shell中等价于"NOP"（即no op，空操作）与shell内建命令true有同样的效果。它本身也是Bash的内建命令之一，返回值是true（0）。

```
:
echo $?   # 返回0
```

在无限循环中的应用：

```
while :
do
   operation-1
   operation-2
   ...
   operation-n
done

# 等价于
#    while true
#    do
#      ...
#    done
```

可在 `if/then` 中充当占位符：

```
if condition
then :   # 什么都不做，跳出判断执行下一条语句
else
   take-some-action
fi
```

在二元操作中作占位符: 查看样例 *8-2*或默认参数部分。

```
: ${username=`whoami`}
# ${username=`whoami`}    如果没有:就会报错
#                          除非 "username" 是系统命令或内建命令

: ${1?"Usage: $0 ARGUMENT"}     # 摘自样例脚本 "usage-message.sh"
```

查看样例 *19-10*了解空命令在here document中作为占位符的情况。

使用参数替换为字符串变量赋值（查看样例 *10-7*）。

```
: ${HOSTNAME?} ${USER?} ${MAIL?}
#  如果其中一个或多个必要的环境变量没有被设置
#  将会打印错误
```

查看变量扩展或字符串替换章节了解空命令在其中的作用。

与 `>` 重定向操作符结合，可以在不改变文件权限的情况下清空文件。如果文件不存在，那么将创建这个文件。

```
: > data.xxx   # 文件 "data.xxx" 已被清空

# 与 cat /dev/null >data.xxx 作用相同
# 但是此操作不会产生一个新进程，因为 ":" 是shell内建命令。
```

也可查看样例 *16-15*。

与 `>>` 重定向操作符结合，将不会清空任何已存在的文件（ `: >> target_file` ）。如果文件不存在，将创建这个文件。

>  以上操作仅适用于普通文件，不适用于管道、符号链接和特殊文件。

空命令可以用来作为一行注释的开头，尽管我们并不推荐这么做。使用 # 可以使解释器关闭该行的错误检测，所以几乎所有的内容都可以出现在注释#中。使用空命令却不是这样的：

```
: 这一行注释将会产生一个错误，( if [ $x -eq 3] )。
```

:也可以作为一个域分隔符，比如在 `/etc/passwd` 和 `$PATH` 变量中。

```
bash$ echo $PATH
/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/sbin:/usr/sbin:/usr
/games
```

将冒号作为函数名也是可以的。

```
:()
{
  echo "The name of this function is "$FUNCNAME" "
  # 为什么要使用冒号作函数名？
  # 这是一种混淆代码的方法......
}

:

# 函数名是 :
```

这种写法并不具有可移植性，也不推荐使用。事实上，在Bash的最近的版本更新中已经禁用了这种用法。但我们还可以使用下划线 _ 来替代。

冒号也可以作为非空函数的占位符。

```
not_empty ()
{
  :
} # 含有空指令，这并不是一个空函数。
```

!

取反（或否定）操作符[感叹号]。! 操作符反转已执行的命令的返回状态（查看样例 *6-2*）。它同时可以反转测试操作符的意义，例如可以将相等（=）反转成不等（!=）。它是一个Bash关键词。

在一些特殊场景下，它也会出现在间接变量引用中。

在另外一些特殊场景下，即在命令行下可以使用！调用Bash的历史记录（附录*L*）。需要注意的是，在脚本中，这个机制是被禁用的。

*

通配符[星号]。在文件匹配（globbing）操作时扩展文件名。如果它独立出现，则匹配该目录下的所有文件。

```
bash$ echo *
abs-book.sgml add-drive.sh agram.sh alias.sh
```

在正则表达式中表示匹配任意多个（包括0）前个字符。

*

算术运算符。在进行算术运算时，表示乘法运算。

** 双星号可以表示乘方运算或扩展文件匹配。

?

测试操作符[问号]。在一些特定的语句中，? 表示一个条件测试。

在一个双圆括号结构中，? 可以表示一个类似C语言风格的三元（trinary）运算符的一个组成部分。[2]

`condition?result-if-true:result-if-false`

```
(( var0 = var1<98?9:21 ))
#不要加空格，紧挨着写

#等价于
# if [ "$var1" -lt 98 ]
# then
#   var0=9
# else
#   var0=21
# fi
```

在参数替换表达式中，? 用来测试一个变量是否已经被赋值。

?

通配符。它在进行文件匹配（globbing）时以单字符通配符扩展文件名。 在扩展正则表达式中匹配一个单字符。

$

取值符号[钱字符]，用来进行变量替换（即取出变量的内容）。

```
var1=5
var2=23skidoo

echo $var1     # 5
echo $var2     # 23skidoo
```

如果在变量名前有 $，则表示此变量的值。

$

行结束符[EOF]。 在正则表达式中，$ 匹配行尾字符串。

${}

参数替换。

$'...'

引用字符串扩展。这个结构将转义八进制或十六进制的值转换成ASCII[3]或Unicode字符。

$*, $@

位置参数。

$?

返回状态变量。此变量保存一个命令、一个函数或该脚本自身的返回状态。

$$

进程ID变量。此变量保存该运行脚本的进程ID[4]。

()

命令组。

```
(a=hello; echo $a)
```

❗ 通过括号执行一系列命令会产生一个子shell（subshell）。 括号中的变量，即在子shell中的变量，在脚本的其他部分是不可见的。父进程脚本不能访问子进程（子shell）所创建的变量。

```
a=123
( a=321; )

echo "a = $a"    # a = 123
# 在括号中的 "a" 就像个局部变量。
```

数组初始化。

```
Array=(element1 element2 element3)
```

{xxx,yyy,zzz,...}

花括号扩展结构。

```
echo \"{These,words,are,quoted}\"   # " 将作为单词的前缀和后缀
# "These" "words" "are" "quoted"


cat {file1,file2,file3} > combined_file
# 将 file1, file2 与 file3 拼接在一起后写入 combined_file 中。

cp file22.{txt,backup}
# 将 "file22.txt" 拷贝为 "file22.backup"
```

这个命令可以作用于花括号内由逗号分隔的文件描述列表。[5] 文件名扩展（匹配）作用于大括号间的各个文件。

 除非被引用或被转义，否则空白符不应在花括号中出现。

```
echo {file1,file2}\ :{\ A," B",' C'}
file1 : A file1 : B file1 : C file2 : A file2 : B file2 : C
```

{a..z}

扩展的花括号扩展结构。

```
echo {a..z} #  a b c d e f g h i j k l m n o p q r s t u v w x y
 z
# 输出 a 到 z 之间所有的字母。

echo {0..3} # 0 1 2 3
# 输出 0 到 3 之间所有的数字。


base64_charset=( {A..Z} {a..z} {0..9} + / = )
# 使用扩展花括号初始化一个数组。
# 摘自 vladz 编写的样例脚本 "base64.sh"。
```

Bash第三版中引入了 {a..z} 扩展的花括号扩展结构。



代码块[花括号]，又被称作内联组（inline group）。它实际上创建了一个匿名函数（anonymous function），即没有名字的函数。但是，不同于那些“标准”函数，代码块内的变量在脚本的其他部分仍旧是可见的。

```
bash$ { local a;
              a=123; }
bash: local: can only be used in a
function
```

```
a=123
{ a=321; }
echo "a = $a"   # a = 321   (代码块内赋值)

# 感谢S.C.
```

代码块可以经由I/O重定向进行输入或输出。

样例 **3-1.** 代码块及**I/O**重定向

```
#!/bin/bash
# 读取文件 /etc/fstab

File=/etc/fstab

{
read line1
read line2
} < $File

echo "First line in $File is:"
echo "$line1"
echo
echo "Second line in $File is:"
echo "$line2"

exit 0

# 你知道如何解析剩下的行吗？
# 提示：使用 awk 或...
# Hans-Joerg Diers 建议：使用Bash的内建命令 set。
```

样例 **3-2.** 将代码块的输出保存至文件中

```
#!/bin/bash
# rpm-check.sh

# 查询一个rpm文件的文件描述、包含文件列表，以及是否可以被安装。
# 将输出保存至文件。
#
# 这个脚本使用代码块来描述。

SUCCESS=0
E_NOARGS=65

if [ -z "$1" ]
then
  echo "Usage: `basename $0` rpm-file"
  exit $E_NOARGS
```

```
fi

{ # 代码块起始
  echo
  echo "Archive Description:"
  rpm -qpi $1       # 查询文件描述。
  echo
  echo "Archive Listing:"
  rpm -qpl $1       # 查询文件列表。
  echo
  rpm -i --test $1  # 查询是否可以被安装。
  if [ "$?" -eq $SUCCESS ]
  then
    echo "$1 can be installed."
  else
    echo "$1 cannot be installed."
  fi
  echo              # 代码块结束。
} > "$1.test"       # 输出重定向至文件。

echo "Results of rpm test in file $1.test"

# rpm各项参数的具体含义可查看man文档

exit 0
```

与由圆括号包裹起来的命令组不同，由花括号包裹起来的代码块不产生子进程。[6]

也可以使用非标准的 for 循环语句来遍历代码块。

{}

文本占位符。在 `xargs -i` 后作为输出的占位符使用。

```
ls . | xargs -i -t cp ./{} $1
#            ^^         ^^

# 摘自 "ex42.sh" (copydir.sh)
```

{} \;

路径名。通常在 `find` 命令中使用，但这不是shell的内建命令。

> 定义：路径名是包含完整路径的文件名，例如 `/home/bozo/Notes/Thursday/schedule.txt` 。我们通常又称之为绝对路径。

在执行 `find -exec` 时最后需要加上 `;` ，但是分号需要被转义以保证其不会被shell解释。

[ ]

测试。在 [ ] 之间填写测试表达式。值得注意的是，[是shell内建命令 `test` 的一个组成部分，而不是外部命令 `/usr/bin/test` 的链接。

[[ ]]

测试。在 [[ ]] 之间填写测试表达式。相比起单括号测试（[ ]），它更加的灵活。它是一个shell的关键字。

详情查看关于 *[[ ]]* 结构的讨论。

[ ]

数组元素。在数组中，可以使用中括号的偏移量来用来访问数组中的每一个元素。

```
Array[1]=slot_1
echo ${Array[1]}
```

[ ]

字符集、字符范围。 在正则表达式中，中括号用来匹配指定字符集或字符范围内的任意字符。

$[...]

整数扩展符。在 $[ ] 中可以计算整数的算术表达式。

```
a=3
b=7

echo $[$a+$b]    # 10
echo $[$a*$b]    # 21
```

(( ))

整数扩展符。在 (( )) 中可以计算整数的算术表达式。

详情查看关于 *((...))* 结构的讨论。

> &> >& >> < <>

重定向。

`scriptname >filename` 将脚本 *scriptname* 的输出重定向到 *filename* 中。如果文件存在，那么覆盖掉文件内容。

`command &>filename` 将命令 *command* 的标准输出(stdout) 和标准错误输出(stderr) 重定向到 *filename*。

重定向在用于清除测试条件的输出时特别有效。例如测试一个特定的命令是否存在。

```
bash$ type bogus_command &>/dev/null


bash$ echo $?
1
```

或写在脚本中：

```
command_test () { type "$1" &>/dev/null; }
#                                      ^

cmd=rmdir            # 存在的命令。
command_test $cmd; echo $?   # 返回0


cmd=bogus_command    # 不存在的命令。
command_test $cmd; echo $?   # 返回1
```

`command >&2` 将命令的标准输出重定向至标准错误输出。

`scriptname >>filename` 将脚本 *scriptname* 的输出追加到 *filename* 文件末尾。如果文件不存在，那么将创建这个文件。

`[i]<>filename` 打开文件 *filename* 用来读写，并且分配一个文件描述符*i*指向它。如果文件不存在，那么将创建这个文件。

进程替换： `(command)>` `<(command)`

在某些情况下，"<" 与 ">" 将用作字符串比较。

在另外一些情况下，"<" 与 ">" 将用作数字比较。详情查看样例 *16-9*。

<<

在here document中进行重定向。

<<<

在here string中进行重定向。

<, >

ASCII码比较。

```
veg1=carrots
veg2=tomatoes

if [[ "veg1" < "veg2" ]]
then
  echo "Although $veg1 precede $veg2 in the dictionary,"
  echo -n "this does not necessarily imply anything "
  echo "about my culinary preferences."
else
  echo "What kind of dictionary are you using, anyhow?"
fi
```

\<, >

正则表达式中的单词边界（word boundary）。

```
bash$ grep '\<the\>' textfile
```

|

管道（pipe）。管道可以将上一个命令的输出作为下一个命令的输入，或者直接输出到shell中。管道是一种可以将一系列命令连接在一起的绝妙方式。

```
echo ls -l | sh
#  将 "echo ls -l" 的结果输出到shell中，
#  与直接输入 "ls -l" 的结果相同。



cat *.lst | sort | uniq
# 将所有后缀名为 lst 的文件合并后排序，接着删掉所有重复行。
```

> 管道是一种在进程间通信的典型方法。它将一个进程的输出作为另一个进程的输入。举一个经典的例子，像 `cat` 或者 `echo` 这样的命令，可以通过管道将它们产生的数据流导入到过滤器（filter）中。过滤器是可以用来处理输入流的命令。[7]
>
> `cat $filename1 $filename2 | grep $search_word`
>
> 查看UNIX FAQ第三章获取更多关于使用UNIX管道的信息。

命令的输出同样可以通过管道输入到脚本中。

```
#!/bin/bash
# uppercase.sh : 将所有输入变成大写

tr 'a-z' 'A-Z'
#  为了防止产生单字符文件名，
#  必须使用单引号引用字符范围。

exit 0
```

现在，让我们将 `ls -l` 的输出通过管道导入到脚本中。

```
bash$ ls -l | ./uppercase.sh
 -RW-RW-R--    1 BOZO  BOZO       109 APR  7 19:49 1.TXT
 -RW-RW-R--    1 BOZO  BOZO       109 APR 14 16:48 2.TXT
 -RW-R--R--    1 BOZO  BOZO       725 APR 20 20:56 DATA-FILE
```

在管道中，每一个进程的输出必须作为下个进程的输入被正确读入，如果不这样，数据流会被阻塞（block），管道就无法按照预期正常工作。

```
cat file1 file2 | ls -l | sort
# "cat file1 file2" 的输出会消失。
```

管道是在一个子进程中运行的，因此它并不能修改父进程脚本中的变量。

```
variable="initial_value"
echo "new_value" | read variable
echo "variable = $variable"     # variable = initial_value
```

如果管道中的任意一个命令意外中止了，管道将会提前中断，我们称其为管道破裂(Broken Pipe)。出现这种情况，系统将发送一个 `SIGPIPE` 信号。

>|

强制重定向。即使在 `noclobber` 选项被设置的情况下，重定向也会覆盖已存在的文件。

||

或（OR）逻辑运算符。在测试结构中，任意一个测试条件为真，整个表达式为真。返回 0（成功标志位）。

&

后台运行操作符。如果命令后带&，那么此命令将转至后台运行。

```
bash$ sleep 10 &
[1] 850
[1]+  Done                    sleep 10
```

在脚本中，命令甚至循环都可以在后台运行。

样例 **3-3.** 在后台运行的循环

```
#!/bin/bash
# background-loop.sh

for i in 1 2 3 4 5 6 7 8 9 10            # 第一个循环
do
  echo -n "$i "
done & # 这个循环在后台运行。
```

```
      # 有时会在第二个循环结之后才执行此后台循环。

echo   # 此'echo' 有时不显示

for i in 11 12 13 14 15 16 17 18 19 20   # 第二个循环
do
  echo -n "$i "
done

echo   # 此'echo' 有时不显示

# ======================================================

# 脚本期望输出结果：
# 1 2 3 4 5 6 7 8 9 10
# 11 12 13 14 15 16 17 18 19 20

# 一些情况下可能会输出：
# 11 12 13 14 15 16 17 18 19 20
# 1 2 3 4 5 6 7 8 9 10 bozo $
# 第二个 'echo' 没有被执行，为什么？

# 另外一些情况下可能会输出：
# 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
# 第一个 'echo' 没有被执行，为什么？

# 非常罕见的情况下，可能会输出：
# 11 12 13 1 2 3 4 5 6 7 8 9 10 14 15 16 17 18 19 20
# 前台循环抢占（preempt）了后台循环。

exit 0

#  Nasimuddin Ansari 建议：在第6行和第14行的
#  echo -n "$i " 后增加 sleep 1，
#  会得到许多有趣的结果。
```

 脚本在后台执行命令时可能因为等待键盘事件被挂起。幸运的是，有一套方案可以解决这个问题。

&&

与（AND）逻辑操作符。在测试结构中，所有测试条件都为真，表达式才为真，返回 0（成功标志位）。

-

选项与前缀。它可以作为命令的选项标志，也可以作为一个操作符的前缀，也可以作为在参数代换中作为默认参数的前缀。

`COMMAND -[Option1][Option2][..]`

`ls -al`

`sort -dfu $filename`

```
if [ $file1 -ot $file2 ]
then #      ^
  echo "File $file1 is older than $file2."
fi

if [ "$a" -eq "$b" ]
then #    ^
  echo "$a is equal to $b."
fi

if [ "$c" -eq 24 -a "$d" -eq 47 ]
then #    ^               ^
  echo "$c equals 24 and $d equals 47."
fi


param2=${param1:-$DEFAULTVAL}
#               ^
```

--

双横线一般作为命令长选项的前缀。

`sort --ignore-leading-blanks`

双横线与Bash内建命令一起使用时，意味着该命令选项的结束。

ⓘ 下面提供了一种删除文件名以横线开头文件的简单方法。

```
bash$ ls -l
-rw-r--r-- 1 bozo bozo 0 Nov 25 12:29 -badname


bash$ rm -- -badname

bash$ ls -l
total 0
```

双横线通常也和 `set` 连用。

`set -- $variable` （查看样例 *15-18*）。

-

重定向输入输出[短横线]。

```
bash$ cat -
abc
abc

...

Ctl-D
```

在这个例子中， `cat -` 输出由键盘读入的标准输入(stdin) 到 标准输出(stdout)。但是在真实应用的 I/O 重定向中是否有使用 '-'？

```
(cd /source/directory && tar cf - . ) | (cd /dest/directory && t
ar xpvf -)

# 将整个文件树从一个目录移动到另一个目录。
```

```
# 感谢 Alan Cox <a.cox@swansea.ac.uk> 所作出的部分改动

# 1) cd /source/directory
#    工作目录定位到文件所属的源目录
# 2) &&
#    "与链"：如果 'cd' 命令操作成功，那么执行下一条命令
# 3) tar cf - .
#    'tar c' (create 创建) 创建一份新的档案
#    'tar f -' (file 指定文件) 在 '-' 后指定一个目标文件作为输出
#    '.' 代表当前目录
# 4) |
#    通过管道进行重定向
# 5) ( ... )
#    在建立的子进程中执行命令
# 6) cd /dest/directory
#    工作目录定位到目标目录
# 7) &&
#    与 2) 相同
# 8) tar xpvf -
#    'tar x' 解压档案
#    'tar p' (preserve 保留) 保留档案内文件的所有权及文件权限
#    'tar v' (verbose 冗余) 发送全部信息到标准输出
#    'tar f -' (file 指定文件) 在 '-' 后指定一个目标文件作为输入
#
#    注意 'x' 是一个命令，而 'p', 'v', 'f' 是选项。

# 干的漂亮！



# 更加优雅的写法是:
#   cd source/directory
#   tar cf - . | (cd ../dest/directory; tar xpvf -)
#
# 同样可以写成:
#   cp -a /source/directory/* /dest/directory
# 或:
#   cp -a /source/directory/* /source/directory/.[^.]* /dest/dir
ectory
# 可以在源目录中有隐藏文件时使用
```

```
bunzip2 -c linux-2.6.16.tar.bz2 | tar xvf -
#  --未解压的 tar 文件--         | --将解压出的 tar 传递给 "tar"--
#  如果不使用管道让 "tar" 处理 "bunzip2" 得到的文件，
#  那么就需要使用单独的两步来完成。
#  目的是为了解压 "bzipped" 压缩的内核源代码。
```

下面的例子中，"-" 并不是一个Bash的操作符，它仅仅是 `tar`，`cat` 等一些特定UNIX命令中将结果输出到标准输出的选项。

```
bash$ echo "whatever" | cat -
whatever
```

当需要文件名的时候，**-** 可以用来代替某个文件而重定向到标准输出（通常出现在 `tar cf` 中）或从 *stdin* 中接受数据。这是一种在管道中使用面向文件（file-oriented）工具作为过滤器的方法。

```
bash$ file
Usage: file [-bciknvzL] [-f namefile] [-m magicfiles] file...
```

单独执行 `file` 命令，将会得到一条错误信息。

在命令后增加一个 "-" 可以得到一个更加有用的结果。它会使得shell暂停等待用户输入。

```
bash$ file -
abc
standard input:              ASCII text



bash$ file -
#!/bin/bash
standard input:              Bourne-Again shell script text exec
utable
```

现在命令能够接受标准输入并且处理它们了。

"-" 能够通过管道将标准输出重定向到其他命令中。这就可以做到像在某个文件前添加几行这样的事情。

使用 `diff` 比较两个文件的部分内容：

```
grep Linux file1 | diff file2 -
```

最后介绍一个使用 **-** 的 `tar` 命令的实际案例。

样例 **3-4.** 备份最近一天修改过的所有文件

```
#!/bin/bash

#  将当前目录下24小时之内修改过的所有文件备份成一个
#  "tarball" (经 tar 打包`与 gzip 压缩) 文件

BACKUPFILE=backup-$(date +%m-%d-%Y)
#                 在备份文件中嵌入时间
#                 感谢 Joshua Tschida 提供的建议

archive=${1:-$BACKUPFILE}
#  如果没有在命令行中特别制定备份格式，
#  那么将会默认设置为 "backup-MM-DD-YYYY.tar.gz"。

tar cvf - `find . -mtime -1 -type f -print` > $archive.tar
gzip $archive.tar
echo "Directory $PWD backed up in archive file \"$archive.tar.gz
\"."

#  Stephane Chazeles 指出如果目录中有非常多的文件，
#  或文件名中包含空白符时，上面的代码会运行失败。

# 他建议使用以下的任意一种方法：
# -------------------------------------------------------------
-----
#   find . -mtime -1 -type f -print0 | xargs -0 tar rvf "$archiv
e.tar"
#   使用了 GNU 版本的 "find" 命令。


#   find . -mtime -1 -type f -exec tar rvf "$archive.tar" '{}' \;

#   兼容其他的 UNIX 发行版，但是速度会比较慢
# -------------------------------------------------------------
-----


exit 0
```

⚠️ 以 "-" 开头的文件在和"-" 重定向操作符一起使用时可能会导致一些问题。因此合格的脚本必须首先检查这种情况。如果遇到，就需要给文件名加一个合适的前缀，比如 `./-FILENAME, $PWD/-FILENAME` 或者 `$PATHNAME/-FILENAME` 。

如果变量的值以 '-' 开头，也可能会造成类似问题。

```
var='-n'
echo $var
# 等同于 "echo -n"，不会输出任何东西。
```

-

先前的工作目录。使用 `cd -` 命令可以返回先前的工作目录。它实际上是使用了 `$OLDPWD` 环境变量。

⚠️ 不要将这里的 "-" 与先前的 "-" 重定位操作符混淆。"-" 的具体含义需要根据上下文来解释。

-

减号。算术运算符中的减法标志。

=

等号。赋值操作符。

```
a=28
echo $a   # 28
```

在一些情况下，"=" 可以作为字符串比较操作符。

+

加号。加法算术运算。

在一些情况下，+ 是作为正则表达式中的一个操作符。

+

选项操作符。作为一个命令或过滤器的选项标记。

特定的一些指令和内建命令使用 + 启用特定的选项，使用 - 禁用特定的选项。在参数代换中，+ 是作为变量扩展的备用值（alternate value）的前缀。

%

取模。取模操作运算符。

```
let "z = 5 % 3"
echo $z  # 2
```

在另外一些情况下，% 是一种模式匹配的操作符。

~

主目录[波浪号]。它相当于内部变量 `$HOME`。 `~bozo` 是 bozo 的主目录，执行 `ls ~bozo` 将会列出他的主目录中内容。 `~/` 是当前用户的主目录，执行 `ls ~/` 将会列出其中所有的内容。

```
bash$ echo ~bozo
/home/bozo

bash$ echo ~
/home/bozo

bash$ echo ~/
/home/bozo/

bash$ echo ~:
/home/bozo:

bash$ echo ~nonexistent-user
~nonexistent-user
```

~+

当前工作目录。它等同于内部变量 `$PWD` 。

~-

先前的工作目录。它等同于内部变量 `$OLDPWD` 。

=~

正则表达式匹配。将在 *Bash version 3* 章节中介绍。

^

行起始符。在正则表达式中，"^" 代表一行文本的开始。

^, ^^

参数替换中的大写转换符（在Bash第4版新增）。

控制字符

改变终端或文件显示的一些行为。一个控制符是由 *CONTRL + key* 组成的（同时按下）。控制字符同样可以通过转义以八进制或十六进制的方式显示。

控制符不能在脚本中使用。

Ctrl-A

移动光标至行首。

Ctrl-B

非破坏性退格（即不删除字符）。

Ctrl-C

中断指令。终止当前运行的任务。

Ctrl-D

登出shell（类似 `exit` ）

键入 `EOF` （end-of-file，文件终止标记），中断 *stdin* 的输入。

当你在终端或 *xterm* 窗口中输入字符时， `Ctl-D` 将会删除光标上的字符。当没有字符时， `Crl-D` 将会登出shell。在 *xterm* 中，将会关闭整个窗口。

Ctrl-E

移动光标至行末。

Ctrl-F

光标向前移动一个字符。

Ctrl-G

响铃 `BEL` 。在一些老式打字机终端上，将会响铃。而在 *xterm* 中，将会产生“哔”声 。

Ctrl-H

抹除（破坏性退格）。退格删除前面的字符。

```
#!/bin/bash
# 在字符串中嵌入 Ctrl-H

a="^H^H"                  # 两个退格符 Ctrl-H
                          # 在 vi/vim 中使用 Ctrl-V Ctrl-H 来键入
echo "abcdef"             # abcdef
echo
echo -n "abcdef$a "       # abcd f
#                ^              ^ 末尾有空格退格两次的结果
echo
echo -n "abcdef$a"        # abcdef
#                                ^ 末尾没有空格时为什么退格无效了？
                          # 并不是我们期望的结果。
echo; echo

# Constantin Hagemeier 建议尝试一下：
# a=$'\010\010'
# a=$'\b\b'
# a=$'\x08\x08'
# 但是这些并不会改变结果。

########################################

# 现在来试试这个。

rubout="^H^H^H^H^H"       # 5个 Ctrl-H

echo -n "12345678"
sleep 2
echo -n "$rubout"
sleep 2
```

Ctrl-I

水平制表符。

Ctrl-J

另起一行（换行）。在脚本中，你也可使用八进制 '\012' 或者十六进制 '\x0a' 来表示。

Ctrl-K

垂直制表符。

当你在终端或 *xterm* 窗口中输入字符时，`Ctrl-K` 将会删除光标上及其后的所有字符。而在脚本中，`Ctrl-K` 的作用有些不同。具体查看下方 Lee Lee Maschmeyer 写的样例。

Ctrl-L

清屏、走纸。在终端中等同于 `clear` 命令。在打印时，`Ctrl-L` 将会使纸张移动到底部。

Ctrl-M

回车（CR）。

```
#!/bin/bash
# 感谢 Lee Maschmeyer 提供的样例。

read -n 1 -s -p \
$'Control-M leaves cursor at beginning of this line. Press Enter
. \x0d'
           # '0d' 是 Control-M 的十六进制的值
echo >&2   # '-s' 参数禁用了回显，所以需要显式的另起一行。

read -n 1 -s -p $'Control-J leaves cursor on next line. \x0a'
           # '0a' 是 Control-J 换行符的十六进制的值
echo >&2

###

read -n 1 -s -p $'And Control-K\x0bgoes straight down.'
echo >&2   # Control-K 是垂直制表符。

# 一个更好的垂直制表符例子是：

var=$'\x0aThis is the bottom line\x0bThis is the top line\x0a'
echo "$var"
#  这将会产生与上面的例子类似的结果。但是
echo "$var" | col
#  这却会使得右侧行高于左侧行。
#  这也解释了为什么我们需要在行首和行尾加上换行符
#  来避免显示的混乱。

# Lee Maschmeyer 的解释：
# --------------------------
#  在第一个垂直制表符的例子中，垂直制表符使其
#  在没有回车的情况下向下打印。
#  这在那些不能回退的设备上，例如 Linux 的终端才可以。
#  而垂直制表符的真正目的是向上而非向下。
#  它可以用来在打印机中用来打印上标。
#  col 工具可以用来模拟真实的垂直制表符行为。

exit 0
```

Ctrl-N

在命令行历史记录中调用下一条历史命令[8]。

Ctrl-O

在命令行中另起一行。

Ctrl-P

在命令行历史记录中调用上一条历史命令。

Ctrl-Q

恢复（XON）。

终端恢复读入 *stdin*。

Ctrl-R

在命令行历史记录中进行搜索。

Ctrl-S

挂起（XOFF）。

终端冻结 *stdin*。（可以使用 `Ctrl-Q` 恢复）

Ctrl-T

交换光标所在字符与其前一个字符。

Ctrl-U

删除光标所在字符之前的所有字符。 在一些情况下，不管光标在哪个位置， `Ctrl-U` 都会删除整行文字。

Ctrl-V

输入时，使用 `Ctrl-V` 允许插入控制字符。例如，下面两条语句是等价的：

```
echo -e '\x0a'
echo <Ctl-V><Ctl-J>
```

`Ctrl-V` 在文本编辑器中特别有用。

Ctrl-W

当你在终端或 *xterm* 窗口中输入字符时，`Ctrl-W` 将会删除光标所在字符之前到其最近的空白符之间的所有字符。在一些情况下，`Ctrl-W` 会删除到之前最近的非字母或数字的字符。

Ctrl-X

在一些特定的文本处理程序中，剪切高亮文本并复制到剪贴板（clipboard）。

Ctrl-Y

粘贴之前使用 `Ctrl-U` 或 `Ctrl-W` 删除的文字。

Ctrl-Z

暂停当前运行的任务。

在一些特定的文本处理程序中是替代操作。

在 MSDOS 文件系统中作为 `EOF`（end-of-file，文件终止标记）。

空白符

作为命令或变量之间的分隔符。空白符包含空格、制表符、换行符或它们的任意组合。[9]在一些地方，比如变量赋值时，空白符不应该出现，否则会造成语法错误。

空白行在脚本中不会有任何实际作用，但是可以划分代码，使代码更具可读性。

特殊变量 `$IFS` 是作为一些特定命令的输入域（field）分隔符，默认值为空白符。

> 定义：域是字符串中离散的数据块。使用空白符或者指定的字符（通常由 `$IFS` 决定）来分隔临近域。在一些情况下，域也可以被称作记录（record）。

如果想在字符串或者变量中保留空白符，请引用。

UNIX 过滤器可以使用 POSIX 字符类 `[:space:]` 来寻找和操作空白符。

> 1. 操作符（operator）用来执行表达式（operation）。最常见的例子就是算术运算符+ - * /。在Bash中，操作符和关键字的概念有一些重叠。↩

> 2. 它更被人熟知的名字是三元（ternary）操作符。但是读起来不清晰，而且容易令人混淆。trinary 是一种更加优雅的写法。↩

> 3. 美国信息交换标准代码（American Standard Code for Information Interchange）。这是一套可以由计算机存储和处理的7位（bit）字符（包含字母、数字和一系列有限的符号）编码系统。↩

> 4. 进程标识符（PID），是分配给正在运行进程的唯一数字标识。可以使用 `ps` 命令查看进程的 PID。
> 定义：进程是正在执行的命令或程序，通常也称作任务。↩

> 5. 由shell来执行大括号扩展操作。命令本身是在扩展的基础上进行操作的。↩

> 6. 例外：作为管道的一部分的大括号中的代码块可能会运行在子进程中。
>
> ```
> ls | { read firstline; read secondline; }
> #  错误。大括号中的代码块在子进程中运行，
> #+ 因此 "ls" 命令输出的结果不能传递到代码块中。
> echo "First line is $firstline; second line is
> $secondline"  # 无效。
>
> # 感谢 S.C.
> ```
>
> ↩

> 7. 正如在古代催情剂（philtre）被认为是一种能引发神奇变化的药剂一样，UNIX 中的过滤器（filter）也是有类似的作用的。
> （如果一个程序员做出了一个能够在 Linux 设备上运行的 "love philtre"，那么他将会获得巨大的荣誉。）↩

8

> 8. Bash将之前在命令行中执行过的命令存储在缓存（buffer）中，或者一块内存区域里。可以使用内建命令 `history` 来查看。↩

> 9. 换行符本身也是一个空白符。因此这就是为什么仅仅包含一个换行符的空行也被认为是空白符。↩

第四章 变量与参数

本章目录

- 4.1 变量替换
- 4.2 变量赋值
- 4.3 Bash变量弱类型
- 4.4 特殊变量类型

变量（variable）在编程语言中用来表示数据。它本身只是一个标记，指向数据在计算机内存中的一个或一组地址。

变量通常出现在算术运算，数量操作及字符串解析中。

4.1 变量替换

变量名是其所指向值的一个占位符（placeholder）。引用变量值的过程我们称之为变量替换（variable substitution）。

$

接下来我们仔细区分一下变量名与变量值。如果变量名是 `variable1` ，那么 `$variable1` 就是对变量值的引用。[1]

```
bash$ variable1=23


bash$ echo variable1
variable1

bash$ echo $variable1
23
```

变量仅仅在声明时、赋值时、被删除时（ `unset` ）、被导出时（ `export` ），算术运算中使用双括号结构((...))时或在代表信号时（signal，查看样例 32-5）才不需要有 $ 前缀。赋值可以是使用 =（比如 `var1=27` ），可以是在 `read` 语句中，也可以是在循环的头部（ `for var2 in 1 2 3` ）。

在双引号 `""` 字符串中可以使用变量替换。我们称之为部分引用，有时候也称弱引用。而使用单引号 `''` 引用时，变量只会作为字符串显示，变量替换不会发生。我们称之为全引用，有时也称强引用。更多细节将在第五章讲解。

实际上， `$variable` 这种写法是 `${variable}` 的简化形式。在某些特殊情况下，使用 `$variable` 写法会造成语法错误，使用完整形式会更好（查看章节 10.2）。

样例 4-1. 变量赋值与替换

```
#!/bin/bash
# ex9.sh
```

```
# 变量赋值与替换

a=375
hello=$a
#   ^ ^

#--------------------------------------------------
# 初始化变量时，赋值号 = 的两侧绝不允许有空格出现。
# 如果有空格会发生什么？

#   "VARIABLE =value"
#            ^
#% 脚本将会尝试运行带参数 "=value" 的 "VARIABLE " 命令。

#   "VARIABLE= value"
#             ^
#% 脚本将会尝试运行 "value" 命令，
#+ 同时设置环境变量 "VARIABLE" 为 ""。
#--------------------------------------------------


echo hello    # hello
# 没有引用变量，"hello" 只是一个字符串...

echo $hello   # 375
#    ^          这是变量引用。

echo ${hello} # 375
#               与上面的类似，变量引用。

# 字符串内引用变量
echo "$hello"    # 375
echo "${hello}"  # 375

echo

hello="A B  C   D"
echo $hello   # A B C D
echo "$hello" # A B  C   D
```

```
# 正如我们所见，echo $hello 与 echo "$hello" 的结果不同。
# ====================================
# 字符串内引用变量将会保留变量的空白符。
# ====================================

echo

echo '$hello'  # $hello
#    ^      ^
#  单引号会禁用掉（转义）变量引用，这导致 "$" 将以普通字符形式被解析。

# 注意单双引号字符串引用效果的不同。

hello=    # 将其设置为空值
echo "\$hello (null value) = $hello"      # $hello (null value) =

# 注意
# 将一个变量设置为空与删除(unset)它不同，尽管它们的表现形式相同。

# -----------------------------------------------

# 使用空白符分隔，可以在一行内对多个变量进行赋值。
# 但是这会降低程序的可读性，并且可能会导致部分程序不兼容的问题。

var1=21  var2=22  var3=$V3
echo
echo "var1=$var1   var2=$var2   var3=$var3"

# 在一些老版本的 shell 中这样写可能会有问题。

# -----------------------------------------------

echo; echo

numbers="one two three"
#           ^   ^
other_numbers="1 2 3"
#               ^ ^
# 如果变量中有空白符号，那么必须用引号进行引用。
# other_numbers=1 2 3                  # 出错
```

```
echo "numbers = $numbers"
echo "other_numbers = $other_numbers"  # other_numbers = 1 2 3
# 也可以转义空白符。
mixed_bag=2\ ---\ Whatever
#           ^    ^ 使用 \ 转义空格

echo "$mixed_bag"         # 2 --- Whatever

echo; echo

echo "uninitialized_variable = $uninitialized_variable"
# 未初始化的变量是空值(null表示不含有任何值)。
uninitialized_variable=   # 只声明而不初始化，等同于设为空值。
echo "uninitialized_variable = $uninitialized_variable" # 仍旧为空


uninitialized_variable=23       # 设置变量
unset uninitialized_variable    # 删除变量
echo "uninitialized_variable = $uninitialized_variable"
                                # uninitialized_variable =
                                # 变量值为空
echo

exit 0
```

> ⚠ 一个未被赋值或未初始化的变量拥有空值（null value）。注意：*null*值不等同于*0*。
>
> ```
> if [ -z "$unassigned" ]
> then
>   echo "\$unassigned is NULL."
> fi     # $unassigned is NULL.
> ```
>
> 在赋值前使用变量可能会导致错误。但在算术运算中使用未赋值变量是可行的。
>
> ```
> echo "$uninitialized"                # 空行
> let "uninitialized += 5"             # 加5
> echo "$uninitialized"                # 5
> # 结论：
> # 一个未初始化的变量不含值(null)，但在算术运算中会被作为0处理。
> ```
>
> 也可参考样例 15-23。

> 1. 实际上，变量名是被称作左值（lvalue），意思是出现在赋值表达式的左侧的值，比如 `VARIABLE=23` 。变量值被称作右值（rvalue），意思是出现在赋值表达式右侧的值，比如 `VAR2=$VARIABLE` 。
> 事实上，变量名只是一个引用，一枚指针，指向实际存储数据内存地址的指针。 ↩

4.2 变量赋值

=

赋值操作符（在其前后没有空白符）。

> ⚠ 不要混淆 = 与 -eq，后者用来进行比较而非赋值。
>
> 同时也要注意 = 根据使用场景既可作赋值操作符，也可作比较操作符。

样例 4-2. 变量赋值

```
#!/bin/bash
# 非引用形式变量

echo

# 什么时候变量是非引用形式，即变量名前没有 '$' 符号的呢？
# 当变量在被赋值而不是被引用时。

# 赋值
a=879
echo "The value of \"a\" is $a."

# 使用 'let' 进行赋值
let a=16+5
echo "The value of \"a\" is now $a."

echo

# 在 'for' 循环中赋值（隐式赋值）
echo -n "Values of \"a\" in the loop are: "
for a in 7 8 9 11
do
  echo -n "$a "
done

echo
echo

# 在 'read' 表达式中（另一种赋值形式）
echo -n "Enter \"a\" "
read a
echo "The value of \"a\" is now $a."

echo

exit 0
```

样例 4-3. 奇妙的变量赋值

```
#!/bin/bash

a=23              # 简单形式
echo $a
b=$a
echo $b

# 来我们玩点炫的（命令替换）。

a=`echo Hello!`   # 将 'echo' 命令的结果赋值给 'a'
echo $a
#  注意在命令替换结构中包含感叹号(!)在命令行中使用将会失效，
#+ 因为它将会触发 Bash 的历史(history)机制。
#  在shell脚本内，Bash 的历史机制默认关闭。

a=`ls -l`         # 将 'ls -l' 命令的结果赋值给 'a'
echo $a           # 不带引号引用，将会移除所有的制表符与分行符
echo
echo "$a"         # 引号引用变量将会保留空白符
                  # 查看 "引用" 章节。

exit 0
```

使用 `$(...)` 形式进行赋值（与反引号不同的新形式），与命令替换形式相似。

```
# 摘自 /etc/rc.d/rc.local
R=$(cat /etc/redhat-release)
arch=$(uname -m)
```

4.3 Bash变量是弱类型的

不同于许多其他编程语言，Bash 并不区分变量的类型。本质上说，*Bash* 变量是字符串，但在某些情况下，Bash 允许对变量进行算术运算和比较。决定因素则是变量值是否只含有数字。

样例 4-4. 整数还是字符串？

```
#!/bin/bash
# int-or-string.sh

a=2334                  # 整数。
let "a += 1"
echo "a = $a "          # a = 2335
echo                    # 依旧是整数。


b=${a/23/BB}            # 将 "23" 替换为 "BB"。
                        # $b 变成了字符串。
echo "b = $b"           # b = BB35
declare -i b            # 将其声明为整数并没有什么卵用。
echo "b = $b"           # b = BB35

let "b += 1"            # BB35 + 1
echo "b = $b"           # b = 1
echo                    # Bash 认为字符串的"整数值"为0。

c=BB34
echo "c = $c"           # c = BB34
d=${c/BB/23}            # 将 "BB" 替换为 "23"。
                        # $d 变为了一个整数。
echo "d = $d"           # d = 2334
let "d += 1"            # 2334 + 1
echo "d = $d"           # d = 2335
echo


# 如果是空值会怎样呢？
```

```
e=''                        # ...也可以是 e="" 或 e=
echo "e = $e"               # e =
let "e += 1"                # 空值是否允许进行算术运算？
echo "e = $e"               # e = 1
echo                        # 空值变为了一个整数。

# 如果时未声明的变量呢？
echo "f = $f"               # f =
let "f += 1"                # 是否允许进行算术运算？
echo "f = $f"               # f = 1
echo                        # 未声明变量变为了一个整数。
#
# 然而……
let "f /= $undecl_var"      # 可以除以0么？
#   let: f /= : syntax error: operand expected (error token is "
 ")
# 语法错误！在这里 $undecl_var 并没有被设置为0！
#
# 但是，仍旧……
let "f /= 0"
#   let: f /= 0: division by 0 (error token is "0")
# 预期之中。


# 在执行算术运算时，Bash 通常将其空值的整数值设为0。
# 但是不要做这种事情！
# 因为这可能会导致一些意外的后果。


# 结论：上面的结果都表明 Bash 中的变量是弱类型的。

exit $?
```

弱类型变量有利有弊。它可以使编程更加灵活、更加容易（给与你足够的想象空间）。但它也同样的容易造成一些小错误，容易养成粗心大意的编程习惯。

为了减轻脚本持续跟踪变量类型的负担，Bash 不允许变量声明。

4.4 特殊的变量类型

局部变量

仅在代码块或函数中才可见的变量（参考函数章节的局部变量部分）。

环境变量

会影响用户及shell行为的变量。

>  一般情况下，每一个进程都有自己的“环境”（environment），也就是一组该进程可以访问到的变量。从这个意义上来说，shell表现出与其他进程一样的行为。
>
> 每当shell启动时，都会创建出与其环境对应的shell环境变量。改变或增加shell环境变量会使shell更新其自身的环境。子进程（由父进程执行产生）会继承父进程的环境变量。
>
>  分配给环境变量的空间是有限的。创建过多环境变量或占用空间过大的环境变量有可能会造成问题。
>
> ```
> bash$ eval "`seq 10000 | sed -e 's/.*/export var&=ZZZZZZZZZZ
> ZZZZ/'`"
>
> bash$ du
> bash: /usr/bin/du: Argument list too long
> ```
>
> 注意，上面的"错误"已经在Linux内核版本号为2.6.23的系统中修复了。
>
> （感谢 Stéphane Chazelas 对此问题的解释并提供了上面的例子。）

如果在脚本中设置了环境变量，那么这些环境变量需要被“导出”，也就是通知脚本所在的环境做出相应的更新。这个“导出”操作就是 `export` 命令。

>  脚本只能将变量导出到子进程，即在这个脚本中所调用的命令或程序。在命令行中调用的脚本不能够将变量回传给命令行环境，即子进程不能将变量回传给父进程。
>
> 定义： 子进程（child process）是由另一个进程，即其父进程（parent process）所启动的子程序。

位置参数

从命令行中传递给脚本的参数[1]： `$0, $1, $2, $3 ...` 即命令行参数。

`$0` 代表脚本名称， `$1` 代表第一个参数， `$2` 代表第二个， `$3` 代表第三个，以此类推[2]。在 `$9` 之后的参数必须被包含在大括号中，如 `${10}, ${11}, ${12}` 。

特殊变量 `$*` 与 `$@` 代表所有位置参数。

样例 4-5. 位置参数

```
#!/bin/bash

# 调用脚本时使用至少10个参数，例如
# ./scriptname 1 2 3 4 5 6 7 8 9 10
MINPARAMS=10

echo

echo "The name of this script is \"$0\"."
# 附带 ./ 代表当前目录
echo "The name of this script is \"`basename $0`\"."
# 除去路径信息（查看 'basename'）

echo

if [ -n "$1" ]              # 测试变量是否存在
then
 echo "Parameter #1 is $1"  # 使用引号转义#
fi

if [ -n "$2" ]
```

```
then
 echo "Parameter #2 is $2"
fi

if [ -n "$3" ]
then
 echo "Parameter #3 is $3"
fi

# ...

if [ -n "${10}" ]  # 大于 $9 的参数必须被放在大括号中
then
 echo "Parameter #10 is ${10}"
fi

echo "-----------------------------------"
echo "All the command-line parameters are: "$*""

if [ $# -lt "$MINPARAMS" ]
then
  echo
  echo "This script needs at least $MINPARAMS command-line argum
ents!"
fi

echo

exit 0
```

在位置参数中使用大括号助记符提供了一种非常简单的方式来访问传入脚本的最后一个参数。在其中会使用到间接引用。

```
args=$#             # 传入参数的个数
lastarg=${!args}
# 这是 $args 的一种间接引用方式

# 也可以使用:       lastarg=${!#}             (感谢 Chris Monson.)
# 这是 $# 的一种间接引用方式。
# 注意 lastarg=${!$#} 是无效的。
```

一些脚本能够根据调用时文件名的不同来执行不同的操作。要达到这样的效果，脚本需要检测 `$0` ，也就是调用时的文件名[3]。同时，也必须存在指向这个脚本所有别名的符号链接文件（symbolic links）。详情查看样例 16-2。

> ⓘ 如果一个脚本需要一个命令行参数但是在调用的时候却没有传入，那么这将会造成一个空变量赋值。这通常不是我们想要的。一种避免的方法是，在使用期望的位置参数时候，在赋值语句两侧添加一个额外的字符。

```
variable1_=$1_  # 而不是 variable1=$1
# 使用这种方法可以在没有位置参数的情况下避免产生错误。

critical_argument01=$variable1_

# 多余的字符可以被去掉，就像下面这样：
variable1=${variable1_/_/}
# 仅仅当 $variable1_ 是以下划线开头时候才会有一些副作用。
# 这里使用了我们稍后会介绍的参数替换模板中的一种。
# （将替换模式设为空等价于删除。）

# 更直接的处理方法就是先检测预期的位置参数是否被传入。
if [ -z $1 ]
then
  exit $E_MISSING_POS_PARAM
fi



#  但是，正如 Fabin Kreutz 指出的，
#+ 上面的方法会有一些意想不到的副作用。
#  更好的方法是使用参数替换：
#         ${1:-$DefaultVal}
#  详情查看第十章“操作变量”的第二节“变量替换”。
```

样例 4-6. *wh*, *whois* 域名查询

```
#!/bin/bash
# ex18.sh

# 在下面三个可选的服务器中进行 whois 域名查询：
# ripe.net, cw.net, radb.net

# 将这个脚本重命名为 'wh' 后放在 /usr/local/bin 目录下

# 这个脚本需要进行符号链接：
# ln -s /usr/local/bin/wh /usr/local/bin/wh-ripe
# ln -s /usr/local/bin/wh /usr/local/bin/wh-apnic
# ln -s /usr/local/bin/wh /usr/local/bin/wh-tucows

E_NOARGS=75


if [ -z "$1" ]
then
  echo "Usage: `basename $0` [domain-name]"
  exit $E_NOARGS
fi

# 检查脚本名，访问对应服务器进行查询。
case `basename $0` in    # 也可以写:    case ${0##*/} in
    "wh"       ) whois $1@whois.tucows.com;;
    "wh-ripe"  ) whois $1@whois.ripe.net;;
    "wh-apnic" ) whois $1@whois.apnic.net;;
    "wh-cw"    ) whois $1@whois.cw.net;;
    *          ) echo "Usage: `basename $0` [domain-name]";;
esac

exit $?
```

使用 `shift` 命令可以将全体位置参数向左移一位，重新赋值。

`$1 <--- $2`，`$2 <--- $3`，`$3 <--- $4`，以此类推。

原先的 `$1` 将会消失，而 `$0` （脚本名称）不会有任何改变。如果你在脚本中使用了大量的位置参数， `shift` 可以让你不使用{大括号}助记法也可以访问超过10个的位置参数。

样例 4-7. 使用 `shift` 命令

```
#!/bin/bash
# shft.sh: 使用 `shift` 命令步进访问所有的位置参数。

# 将这个脚本命名为 shft.sh，然后在调用时跟上一些参数。
# 例如：
#    sh shft.sh a b c def 83 barndoor

until [ -z "$1" ]  # 直到访问完所有的参数
do
  echo -n "$1 "
  shift
done

echo               # 换行。

# 那些被访问完的参数又会怎样呢？
echo "$2"
# 什么都不会被打印出来。
# 当 $2 被移动到 $1 且没有 $3 时，$2 将会保持空。
# 因此 shift 是移动参数而非复制参数。

exit

#  可以参考 echo-params.sh 脚本，在不使用 shift 命令的情况下，
#+ 步进访问所有位置参数。
```

`shift` 命令也可以带一个参数来指明一次移动多少位。

```
#!/bin/bash
# shift-past.sh

shift 3    # 移动3位。
# 与 n=3; shift $n 效果相同。

echo "$1"

exit 0

# ======================== #


$ sh shift-past.sh 1 2 3 4 5
4

#  但是就像 Eleni Fragkiadaki 指出的那样，
#  如果尝试将位置参数（$#）传给 'shift'，
#  将会导致脚本错误的结束，同时位置参数也不会发送改变。
#  这也许是因为陷入了一个死循环...
#  比如：
#      until [ -z "$1" ]
#      do
#         echo -n "$1 "
#         shift 20    #  如果少于20个位置参数，
#      done           #+ 那么循环将永远不会结束。
#
#  当你不确定是否有这么多的参数时，你可以加入一个测试：
#      shift 20 || break
#               ^^^^^^^^
```

>  使用 `shift` 命令同给函数传参相类似。详情查看样例 36-18。

> 1. 函数同样也可以接受与使用位置参数。 ↩

2

> 2. 是调用脚本的进程设置了 $0 参数。就是脚本的文件名。详情可以查看 `execv` 的使用手册。
> 在命令行中，$0 是shell的名称。
>
> ```
> bash$ echo $0
> bash
>
> tcsh% echo $0
> tcsh
> ```
>
> ↩

> 3. 如果脚本被引用（sourced）执行或者被链接（symlinked）执行时会失效。安全的方法是检测变量 `$BASH_Source` 。↩

第五章 引用

本章目录

- 5.1 引用变量
- 5.2 转义

引用就是将一个字符串用引号括起来。这样做是为了保护Shell/Shell脚本中被重新解释过或带扩展功能的特殊字符（如果一个字符带有其特殊意义而不仅仅是字面量的话，这个字符就能称为“特殊字符”。比如星号“*”就能表示正则表达式中的一个通配符）。

```
bash$ ls -l [Vv]*
-rw-rw-r--    1 bozo  bozo       324 Apr  2 15:05 VIEWDATA.BAT
-rw-rw-r--    1 bozo  bozo       507 May  4 14:25 vartrace.sh
-rw-rw-r--    1 bozo  bozo       539 Apr 14 17:11 viewdata.sh

bash$ ls -l '[Vv]*'
ls: [Vv]*: No such file or directory
```

> 可以看到，提示不存在该文件。这里的 `'[Vv]*` 被当成了文件名。 在日常沟通和写作中，当我们引用一个短语的时候，我们会将它单独隔开并赋予它特殊的意义，而在bash脚本中，当我们引用一个字符串，意味着保留它的字面量。

很多程序和公用代码会展开被引用字符串中的特殊字符。引用的一个重用用途是保护Shell中的命令行参数，但仍然允许调用的程序扩展它。

```
bash$ grep '[Ff]irst' *.txt
file1.txt:This is the first line of file1.txt.
file2.txt:This is the First line of file2.txt.
```

> 在所有.txt文件中找出包含first或者First字符串的行

注意，不加引号的 `grep [Ff]irst *.txt` 在Bash下也同样有效。[1]

引用也可以控制echo命令的断行符。

```
bash$ echo $(ls -l)
total 8 -rw-rw-r-- 1 bo bo 13 Aug 21 12:57 t.sh -rw-rw-r-- 1 bo
bo 78 Aug 21 12:57 u.sh


bash$ echo "$(ls -l)"
total 8
 -rw-rw-r--  1 bo bo  13 Aug 21 12:57 t.sh
 -rw-rw-r--  1 bo bo  78 Aug 21 12:57 u.sh
```

> 1. 前提是当前目录下有文件名为First或first的文件。这也是使用引用的另一个原因。（感谢 Harald Koenig 指出了这一点） ↩

5.1 引用变量

引用变量时，通常建议将变量包含在双引号中。因为这样可以防止除 `$`，`` ` ``（反引号）和 `\`（转义符）之外的其他特殊字符被重新解释。[1]在双引号中仍然可以使用 `$` 引用变量（ `"$variable"` ），也就是将变量名替换为变量值（详情查看样例 4-1）。

使用双引号可以防止字符串被分割。[2]即使参数中拥有很多空白分隔符，被包在双引号中后依旧是算作单一字符。

```
List="one two three"

for a in $List     # 空白符将变量分成几个部分。
do
  echo "$a"
done
# one
# two
# three

echo "---"

for a in "$List"   # 在单一变量中保留所有空格。
do #     ^     ^
  echo "$a"
done
# one two three
```

下面是一个更加复杂的例子：

```
variable1="a variable containing five words"
COMMAND This is $variable1    # 带上7个参数执行COMMAND命令：
# "This" "is" "a" "variable" "containing" "five" "words"

COMMAND "This is $variable1"  # 带上1个参数执行COMMAND命令：
# "This is a variable containing five words"


variable2=""    # 空值。

COMMAND  $variable2 $variable2 $variable2
                # 不带参数执行COMMAND命令。
COMMAND "$variable2" "$variable2" "$variable2"
                # 带上3个参数执行COMMAND命令。
COMMAND "$variable2 $variable2 $variable2"
                # 带上1个参数执行COMMAND命令（2空格）。

# 感谢 Stéphane Chazelas。
```

> ⓘ 当字符分割或者保留空白符出现问题时，才需要在 `echo` 语句中用双引号包住参数。

样例 5-1. 输出一些奇怪的变量

```
#!/bin/bash
# weirdvars.sh: 输出一些奇怪的变量

echo

var="'(]\\{}\$\""
echo $var        # '(]\{}$"
echo "$var"      # '(]\{}$"     没有任何区别。

echo

IFS='\'
echo $var        # '(] {}$"     \ 被转换成了空格，为什么？
echo "$var"      # '(]\{}$"
```

```
# 上面的例子由 Stephane Chazelas 提供。

echo

var2="\\\\\""
echo $var2       #   "
echo "$var2"     # \\"
echo
# 但是...var2="\\\\"" 不是合法的语句，为什么？
var3='\\\\'
echo "$var3"     # \\\\
# 强引用是可以的。


# ************************************************************ #
# 就像第一个例子展示的那样，嵌套引用是允许的。

echo "$(echo '"')"           # "
#    ^           ^


# 在有些时候这种方法非常有用。

var1="Two bits"
echo "\$var1 = "$var1""      # $var1 = Two bits
#    ^                ^

# 或者，可以像 Chris Hiestand 指出的那样：

if [[ "$(du "$My_File1")" -gt "$(du "$My_File2")" ]]
#     ^     ^         ^ ^     ^     ^         ^ ^
then
  ...
fi
# ************************************************************ #
```

单引号（''）与双引号类似，但是在单引号中不能引用变量，因为 `$` 不再具有特殊含义。在单引号中，除 `'` 之外的所有特殊字符都将会被直接按照字面意思解释。可以认为单引号（“全引用”）是双引号（“部分引用”）的一种更严格的形式。

>  因为在单引号中转义符（\）都已经按照字面意思解释了，因此尝试在单引号中包含单引号将不会产生你所预期的结果。
>
> ```
> echo "Why can't I write 's between single quotes"
>
> echo
>
> # 可以采取迂回的方式。
> echo 'Why can'\''t I write '"'"'s between single quotes'
> #    |-------|  |----------|   |-----------------------|
> # 由三个单引号引用的字符串，再加上转义以及双引号包住的单引号组成。
>
> # 感谢 Stéphane Chazelas 提供的例子。
> ```

1

1. 在命令行里，如果双引号包含了 "!" 将会产生错误。这是因为shell将其解释为查看历史命令。而在脚本中，因为历史机制已经被关闭，所以不会产生这个问题。
我们更加需要注意的是在双引号中 `\` 的反常行为，尤其是在使用 `echo -e` 命令时。

```
bash$ echo hello\\!
hello!
bash$ echo "hello\\!"
hello\\!


bash$ echo \\
>
bash$ echo "\\"
>
bash$ echo \a
a
bash$ echo "\a"
\a


bash$ echo x\ty
xty
bash$ echo "x\ty"
x\ty

bash$ echo -e x\ty
xty
bash$ echo -e "x\ty"
x       y
```

在 `echo` 后的双引号中一般会转义 `\` 。并且 `echo -e` 会将 `"\t"` 解释成制表符。
（感谢 Wayne Pollock 提出这些；感谢Geoff Lee 与 Daniel Barclay 对此做出的解释。） ↩

2

> 2. 字符分割（word splitting）在本文中的意思是指将一个字符串分割成独立的、离散的变量。↩

5.2 转义

转义是一种引用单字符的方法。通过在特殊字符前加上转义符 `\` 来告诉shell按照字面意思去解释这个字符。

>  需要注意的是，在一些特定的命令和工具，比如 `echo` 和 `sed` 中，转义字符通常会起到相反的效果，即可能会使得那些字符产生特殊含义。

在 `echo` 与 `sed` 命令中，转义字符的特殊含义

\n

换行（line feed）。

\r

回车（carriage return）。

\t

水平制表符。

\v

垂直制表符。

\b

退格。

\a

警报、响铃或闪烁。

\0xx

ASCII码的八进制形式，等价于 `0nn`，其中 `nn` 是数字。

>  在 `$' ... '` 字符串扩展结构中可以通过转义八进制或十六进制的ASCII码形式给变量赋值，比如 `quote=$'\042'`。

样例 5-2. 转义字符

```
#!/bin/bash
# escaped.sh: 转义字符

#############################################
### 首先让我们先看一下转义字符的基本用法。 ###
#############################################

# 转义新的一行。
# ------------

echo ""

echo "This will print
as two lines."
# This will print
# as two lines.

echo "This will print \
as one line."
# This will print as one line.

echo; echo

echo "============="


echo "\v\v\v\v"      # 按字面意思打印 \v\v\v\v
# 使用 echo 命令的 -e 选项来打印转义字符。
echo "============="
echo "VERTICAL TABS"
echo -e "\v\v\v\v"   # 打印四个垂直制表符。
```

```
echo "=============="

echo "QUOTATION MARK"
echo -e "\042"        # 打印 " （引号，八进制ASCII码为42）。
echo "=============="



# 使用 $'\X' 这样的形式后可以不需要加 -e 选项。

echo; echo "NEWLINE and (maybe) BEEP"
echo $'\n'            # 新的一行。
echo $'\a'            # 警报（响铃）。
                      # 根据不同的终端版本，也可能是闪屏。

# 我们之前介绍了 $'\nnn' 字符串扩展，而现在我们要看到的是...

# ============================================ #
# 自 Bash 第二个版本开始的 $'\nnn' 字符串扩展结构。
# ============================================ #

echo "Introducing the \$\' ... \' string-expansion construct . .
 . "
echo ". . . featuring more quotation marks."

echo $'\t \042 \t'    # 在制表符之间的引号。
# 需要注意的是 '\nnn' 是一个八进制的值。

# 字符串扩展同样适用于十六进制的值，格式是 $'\xhhh'。
echo $'\t \x22 \t'  # 在制表符之间的引号。
# 感谢 Greg Keraunen 指出这些。
# 在早期的 Bash 版本中允许使用 '\x022' 这样的形式。

echo



# 将 ASCII 码字符赋值给变量。
# ----------------------
quote=$'\042'         # 将 " 赋值给变量。
echo "$quote Quoted string $quote and this lies outside the quot
```

```
es."

echo

# 连接多个 ASCII 码字符给变量。
triple_underline=$'\137\137\137'  # 137是 '_' ASCII码的八进制形式
echo "$triple_underline UNDERLINE $triple_underline"

echo

ABC=$'\101\102\103\010'           # 101，102，103是 A, B, C
                                  # ASCII码的八进制形式。
echo $ABC

echo

escape=$'\033'                    # 033 是 ESC 的八进制形式
echo "\"escape\" echoes an $escape"
                                  # 没有可见输出


echo

exit 0
```

下面是一个更加复杂的例子：

样例 5-3. 检测键盘输入

```
#!/bin/bash
# 作者：Sigurd Solaas，作于2011年4月20日
# 授权在《高级Bash脚本编程指南》中使用。
# 需要 Bash 版本高于4.2。

key="no value yet"
while true; do
  clear
  echo "Bash Extra Keys Demo. Keys to try:"
  echo
  echo "* Insert, Delete, Home, End, Page_Up and Page_Down"
  echo "* The four arrow keys"
```

```
echo "* Tab, enter, escape, and space key"
echo "* The letter and number keys, etc."
echo
echo "    d = show date/time"
echo "    q = quit"
echo "================================="
echo

# 将独立的Home键值转换为数字7上的Home键值：
if [ "$key" = $'\x1b\x4f\x48' ]; then
 key=$'\x1b\x5b\x31\x7e'
 #   引用字符扩展结构。
fi

# 将独立的End键值转换为数字1上的End键值：
if [ "$key" = $'\x1b\x4f\x46' ]; then
 key=$'\x1b\x5b\x34\x7e'
fi

case "$key" in
 $'\x1b\x5b\x32\x7e')  # 插入
  echo Insert Key
 ;;
 $'\x1b\x5b\x33\x7e')  # 删除
  echo Delete Key
 ;;
 $'\x1b\x5b\x31\x7e')  # 数字7上的Home键
  echo Home Key
 ;;
 $'\x1b\x5b\x34\x7e')  # 数字1上的End键
  echo End Key
 ;;
 $'\x1b\x5b\x35\x7e')  # 上翻页
  echo Page_Up
 ;;
 $'\x1b\x5b\x36\x7e')  # 下翻页
  echo Page_Down
 ;;
 $'\x1b\x5b\x41')  # 上箭头
  echo Up arrow
```

```
    ;;
    $'\x1b\x5b\x42')  # 下箭头
     echo Down arrow
    ;;
    $'\x1b\x5b\x43')  # 右箭头
     echo Right arrow
    ;;
    $'\x1b\x5b\x44')  # 左箭头
     echo Left arrow
    ;;
    $'\x09')  # 制表符
     echo Tab Key
    ;;
    $'\x0a')  # 回车
     echo Enter Key
    ;;
    $'\x1b')  # ESC
     echo Escape Key
    ;;
    $'\x20')  # 空格
     echo Space Key
    ;;
    d)
     date
    ;;
    q)
     echo Time to quit...
     echo
     exit 0
    ;;
    *)
     echo Your pressed: \'"$key"\'
    ;;
   esac

   echo
   echo "================================="

   unset K1 K2 K3
   read -s -N1 -p "Press a key: "
```

```
  K1="$REPLY"
  read -s -N2 -t 0.001
  K2="$REPLY"
  read -s -N1 -t 0.001
  K3="$REPLY"
  key="$K1$K2$K3"

done

exit $?
```

还可以查看样例 37-1。

\"

转义引号，指代自身。

```
echo "Hello"                      # Hello
echo "\"Hello\" ... he said."     # "Hello" ... he said.
```

\$

转义美元符号（跟在 `\\$` 后的变量名将不会被引用）。

```
echo "\$variable01"            # $variable01
echo "The book cost \$7.98."   # The book cost $7.98.
```

\\

转义反斜杠，指代自身。

```
echo "\\"  # 结果是 \

# 然而...

echo "\"    # 在命令行中会出现第二行并提示输入。
            # 在脚本中会出错。

# 但是...

echo '\'    # 结果是 \
```

>  根据转义符所在的上下文（强引用、弱引用，命令替换或者在 here document）的不同，它的行为也会有所不同。

```
                      #  简单转义与引用
echo \z               #  z
echo \\z              # \z
echo '\z'             # \z
ehco '\\z'            # \\z
echo "\z"             # \z
echo "\\z"            # \z

                      #  命令替换
echo `echo \z`        #  z
echo `echo \\z`       #  z
echo `echo \\\z`      # \z
echo `echo \\\\z`     # \z
echo `echo \\\\\\z`   # \z
echo `echo \\\\\\\z`  # \\z
echo `echo "\z"`      # \z
echo `echo "\\z"`     # \z

                      # Here Document
cat <<EOF
\z
EOF                   # \z

cat <<EOF
\\z
EOF                   # \z

# 以上例子由 Stéphane Chazelas 提供。
```

含有转义字符的字符串可以赋值给变量，但是仅仅将单一的转义符赋值给变量是不可行的。

```
variable=\
echo "$variable"
# 这样做会报如下错误：
# tesh.sh: : command not found
# 单独的转义符不能够赋值给变量。
#
#  事实上，"\" 转义了换行，实际效果是：
#+ variable=echo "$variable"
#+ 这是一个非法的赋值方式。

variable=\
23skidoo
echo "$variable"        # 23skidoo
                        # 因为第二行是一个合法的赋值，因此不会报错。


variable=\
#        \^    转义符后有一个空格
echo "$variable"        # 空格

variable=\\
echo "$variable"        # \

variable=\\\
echo "$variable"
# 这样做会报如下错误：
# tesh.sh: \: command not found
#
#  第一个转义符转转义了第二个，但是第三个转义符仍旧转义的是换行，
#+ 跟开始的那个例子一样，因此会报错。

variable=\\\\
echo "$variable"        # \\
                        # 第二个和第四个转义符被转义了，因此可行。
```

转义空格能够避免在命令参数列表中的字符分割问题。

```
file_list="/bin/cat /bin/gzip /bin/more /usr/bin/less /usr/bin/e
macs-20.7"
# 将一系列文件作为命令的参数。

# 增加两个文件到列表中，并且列出整个表。
ls -l /usr/X11R6/bin/xsetroot /sbin/dump $file_list

echo "-----------------------------------------------------------
---------------"

# 如果我们转义了这些空格会怎样？
ls -l /usr/X11R6/bin/xsetroot\ /sbin/dump\ $file_list
# 错误：因为转义了两个空格，因此前三个文件被连接成了一个参数传递给了 'ls -l'
```

转义符也提供一种可以撰写多行命令的方式。通常，每一行是一个命令，但是转义换行后命令就可以在下一行继续撰写。

```
(cd /source/directory && tar cf - . ) | \
(cd /dest/directory && tar xpvf -)
# 回顾 Alan Cox 的目录树拷贝命令，但是把它拆成了两行。

# 或者你也可以：
tar cf - -C /source/directory . |
tar xpvf - -C /dest/directory
# 可以看下方的注释。
# （感谢 Stéphane Chazelas。）
```

>  在脚本中，如果以 "|" 管道作为一行的结束字符，那么不需要加转义符 \ 也可以写多行命令。但是一个好的编程习惯就是在写多行命令的事后，无论什么情况都要在行尾加上转义符 \。

```
echo "foo
bar"
#foo
#bar

echo

echo 'foo
bar'    # 没有区别。
#foo
#bar

echo

echo foo\
bar     # 转义换行。
#foobar

echo

echo "foo\
bar"     # 没有区别，在弱引用中，\ 转义符仍旧转义了换行。
#foobar

echo

echo 'foo\
bar'     # 在强引用中，\ 就按照字面意思来解释了。
#foo\
#bar

# 由 Stéphane Chazelas 提供的例子。
```

第六章 退出与退出状态

> Bourne shell里存在不明确之处，但人们也会使用它们。
>
> —— Chat Ramey

跟C程序类似，`exit` 命令被用来结束脚本。同时，它也会返回一个值，返回值可以被交给父进程。

每个命令都会返回一个退出状态（exit status），有时也叫做返回状态（return status）或退出码（exit code）。命令执行成功返回0，如果返回一个非0值，通常情况下会被认为是一个错误代码。一个运行状态良好的UNIX命令、程序和工具在正常执行退出后都会返回一个0的退出码，当然也有例外。

同样地，脚本中的函数和脚本本身也会返回一个退出状态。在脚本或者脚本函数中执行的最后的命令会决定它们的退出状态。在脚本中，`exit nnn` 命令将会把nnn退出状态码传递给shell（nnn 必须是 0-255 之间的整型数）。

>  当一个脚本以不带参数的 `exit` 来结束时，脚本的退出状态由脚本最后执行命令决定（`exit` 命令之前）。

```
#!/bin/bash

COMMAND_1

...

COMMAND_LAST

# 将以最后的命令来决定退出状态

exit
```

> `exit`，`exit $?` 以及省略 `exit` 效果等同。

```
#!/bin/bash

COMMAND_1

...

COMMAND_LAST

#将以最后的命令来决定退出状态

exit $?
```

```
#!/bin/bash

COMMAND_1

...

COMMAND_LAST

#将以最后的命令来决定退出状态
```

`$?` 读取上一个执行命令的退出状态。在一个函数返回后， `$?` 给出函数最后执行的那条命令的退出状态。这就是Bash函数的"返回值"。[1]

在管道执行后， `$?` 给出最后执行的那条命令的退出状态。

在脚本终止后，命令行下键入 `$?` 会给出脚本的退出状态，即在脚本中最后一条命令执行后的退出状态。一般情况下，0为成功，1-255为失败。

样例 6-1. 退出与退出状态

```
#!/bin/bash

echo hello
echo $?    # 返回值为0，因为执行成功。

lskdf      # 不认识的命令。
echo $?    # 返回非0值，因为失败了。

echo

exit 113   # 将返回113给shell
           # 为了验证这些，在脚本结束的地方使用“echo $?”

#  按照惯例，'exit 0' 意味着执行成功，
#+ 非0意味着错误或者异常情况。
#  查看附录章节“退出码的特殊含义”
```

`$?` 对于测试脚本中的命令的执行结果特别有用（查看样例 16-35和样例 16-20）。

>  逻辑非操作符 ! 将会反转测试或命令的结果，并且这将会影响退出状态。

样例 6-2. 否定一个条件使用!

```
true    # true 是 shell 内建命令。
echo "exit status of \"true\" = $?"     # 0

! true
echo "exit status of \"! true\" = $?"   # 1
# 注意在命令之间的 "!" 需要一个空格。
# !true 将导致一个"command not found"错误。
#
# 如果一个命令以'!'开头，那么将调用 Bash 的历史机制，显示这个命令被使用的历
史。

true
!true
# 这次就没有错误了，但是同样也没有反转。
# 它不过是重复之前的命令（true）。


# =============================================================== #
# 在 _pipe_ 前使用 ! 将改变返回的退出状态。
ls | bogus_command      #bash: bogus_command: command not found
echo $?                 #127
>
! ls | bogus_command    #bash: bogus_command:command not found
echo $?                 #0
# 注意 ! 不会改变管道的执行。
# 只改变退出状态。
#==============================================================  #
>
# 感谢 Stéphane Chazelas 和 Kristopher Newsome。
```

>  某些特定的退出码具有一些特定的保留含义，用户不应该在自己的脚本中重新定义它们。

> 1. 在函数没有用return来结束这个函数的情况下。↩

第七章 测试

本章目录

- 7.1 测试结构
- 7.2 文件测试操作
- 7.3 其他比较操作
- 7.4 嵌套 if/then 条件测试
- 7.5 牛刀小试

每一个完备的程序设计语言都可以对一个条件进行判断，然后根据判断结果执行相应的指令。Bash 拥有 `test` 命令，双方括号、双圆括号 测试操作符以及 `if/then` 测试结构。

7.1 测试结构

- `if/then` 结构是用来检测一系列命令的 退出状态 是否为0（按 UNIX 惯例，退出码 0 表示命令执行成功），如果为0，则执行接下来的一个或多个命令。
- 测试结构会使用一个特殊的命令 `[` （参看特殊字符章节 左方括号）。等同于 `test` 命令，它是一个内建命令，写法更加简洁高效。该命令将其参数视为比较表达式或文件测试，以比较结果作为其退出状态码返回（0 为真，1 为假）。
- Bash 在 2.02 版本中引入了扩展测试命令 `[[...]]`，它提供了一种与其他语言语法更为相似的方式进行比较操作。注意， `[[` 是一个 关键字 而非一个命令。

  Bash 将 `[[ $a -lt $b ]]` 视为一整条语句，执行并返回退出状态。

- 结构 `(( ... ))` 和 `let ...` 根据其执行的算术表达式的结果决定退出状态码。这样的 算术扩展 结构可以用来进行 数值比较。

```
(( 0 && 1 ))                 # 逻辑与
echo $?     # 1     ***
# 然后 ...
let "num = (( 0 && 1 ))"
echo $num   # 0
# 然而 ...
let "num = (( 0 && 1 ))"
echo $?     # 1     ***


(( 200 || 11 ))              # 逻辑或
echo $?     # 0     ***
# ...
let "num = (( 200 || 11 ))"
echo $num   # 1
let "num = (( 200 || 11 ))"
echo $?     # 0     ***


(( 200 | 11 ))               # 按位或
echo $?                      # 0     ***
# ...
let "num = (( 200 | 11 ))"
echo $num                    # 203
let "num = (( 200 | 11 ))"
echo $?                      # 0     ***

# "let" 结构的退出状态与双括号算术扩展的退出状态是相同的。
```

注意，双括号算术扩展表达式的退出状态码不是一个错误的值。算术表达式为0，返回1；算术表达式不为0，返回0。

```
var=-2 && (( var+=2 ))
echo $?                    # 1

var=-2 && (( var+=2 )) && echo $var
                           # 并不会输出 $var, 因为((var+=2))的状态码
为1
```

- `if` 不仅可以用来测试括号内的条件表达式，还可以用来测试其他任何命令。

```
if cmp a b &> /dev/null  # 消去输出结果
then echo "Files a and b are identical."
else echo "Files a and b differ."
fi

# 下面介绍一个非常实用的 “if-grep" 结构：
# -----------------------------------
if grep -q Bash file
  then echo "File contains at least one occurrence of Bash."
fi

word=Linux
letter_sequence=inu
if echo "$word" | grep -q "$letter_sequence"
# 使用 -q 选项消去 grep 的输出结果
then
  echo "$letter_sequence found in "$word"
else
  echo "$letter_sequence not found in $word"
fi


if COMMAND_WHOSE_EXIT_STATUS_IS_0_UNLESS_ERROR_OCCURRED
  then echo "Command succeed."
  else echo "Command failed."
fi
```

- 感谢 Stéphane Chazelas 提供了后两个例子。

样例 7-1. 什么才是真？

```
#!/bin/bash

# 提示：
# 如果你不确定某个表达式的布尔值，可以用 if 结构进行测试。
```

```
echo

echo "Testing \"0\""
if [ 0 ]
then
  echo "0 is true."
else
  echo "0 is false."
fi              # 0 为真。

echo

echo "Testing \"1\""
if [ 1 ]
then
  echo "1 is true."
else
  echo "1 is false."
fi              # 1 为真。

echo

echo "Testing \"-1\""
if [ -1 ]
then
  echo "-1 is true."
else
  echo "-1 is false."
fi              # -1 为真。

echo

echo "Testing \"NULL\""
if [ ]          # NULL, 空
then
  echo "NULL is true."
else
  echo "NULL is false."
fi              # NULL 为假。
```

```
echo

echo "Testing \"xyz\""
if [ xyz ]     # 字符串
then
  echo "Random string is true."
else
  echo "Random string is false."
fi             # 随机字符串为真。

echo

echo "Testing \"$xyz\""
if [ $xyz ]    # 原意是测试 $xyz 是否为空，但是
               # 现在 $xyz 只是一个没有初始化的变量。
then
  echo "Uninitialized variable is true."
else
  echo "Uninitialized variable is flase."
fi             # 未初始化变量含有null空值，为假。

echo

echo "Testing \"-n \$xyz\""
if [ -n "$xyz" ]            # 更加准确的写法。
then
  echo "Uninitialized variable is true."
else
  echo "Uninitialized variable is false."
fi             # 未初始化变量为假。

echo


xyz=           # 初始化为空。

echo "Testing \"-n \$xyz\""
if [ -n "$xyz" ]
then
  echo "Null variable is true."
```

```
else
  echo "Null variable is false."
fi              # 空变量为假。

echo

# 什么时候 "false" 为真？

echo "Testing \"false\""
if [ "false" ]              #  看起来 "false" 只是一个字符串
then
  echo "\"false\" is true." #+ 测试结果为真。
else
  echo "\"false\" is false."
fi              # "false" 为真。

echo

echo "Testing \"\$false\""  # 未初始化的变量。
if [ "$false" ]
then
  echo "\"\$false\" is true."
else
  echo "\"\$false\" is false."
fi              # "$false" 为假。
                # 得到了我们想要的结果。

# 如果测试空变量 "$true" 会有什么样的结果？

echo

exit 0
```

练习：理解 样例 7-1

```
if [ condition-true ]
then
   command 1
   command 2
   ...
else  # 如果测试条件为假，则执行 else 后面的代码段
   command 3
   command 4
   ...
fi
```

如果把 `if` 和 `then` 写在同一行时，则必须在 `if` 语句后加上一个分号来结束语句。因为 `if` 和 `then` 都是关键字。以关键字（或者命令）开头的语句，必须先结束该语句(分号;)，才能执行下一条语句。

```
if [ -x "$filename" ]; then
```

Else if 与 elif

elif

`elif` 是 `else if` 的缩写。可以把多个 `if/then` 语句连到外边去，更加简洁明了。

```
if [ condition1 ]
then
   command1
   command2
   command3
elif [condition2 ]
# 等价于 else if
then
   command4
   command5
else
   default-command
fi
```

`if test condition-true` 完全等价于 `if [ condition-true ]` 。当语句开始执行时，左括号 `[` 是作为调用 `test` 命令的标记[1]，而右括号则不严格要求，但在新版本的 Bash 里，右括号必须补上。

`test` 命令是 Bash 的 内建命令，可以用来检测文件类型和比较字符串。在 Bash 脚本中， `test` 不调用 `sh-utils` 包下的文件 `/usr/bin/test` 。同样， `[` 也不会调用链接到 `/usr/bin/test` 的 `/usr/bin/[` 文件。

```
bash$ type test
test is a shell builtin
bash$ type '['
[ is a shell builtin
bash$ type '[['
[[ is a shell keyword
bash$ type ']]'
]] is a shell keyword
bash$ type ']'
bash: type: ]: not found
```

如果你想在 Bash 脚本中使用 `/usr/bin/test` ，那你必须把路径写全。

样例 7-2. `test` ， `/usr/bin/test` ， `[]` 和 `/usr/bin/[` 的等价性

```
#!/bin/bash

echo

if test -z "$1"
then
  echo "No command-line arguments."
else
  echo "First command-line argument is $1."
fi

echo

if /usr/bin/test -z "$1"      # 等价于内建命令 "test"
#  ^^^^^^^^^^^^^              # 指定全路径
then
```

```
  echo "No command-line arguments."
else
  echo "First command-line argument is $1."
fi

echo

if [ -z "$1" ]                # 功能和上面的代码相同。
#   if [ -z "$1"                理论上可行，但是 Bash 会提示缺失右括号

then
  echo "No command-line arguments."
else
  echo "First command-line argument is $1."
fi

echo


if /usr/bin/[ -z "$1" ]       # 功能和上面的代码相同。
# if /usr/bin/[ -z "$1"       # 理论上可行，但是会报错
#                             # 已经在 Bash 3.x 版本被修复了
then
  echo "No command-line arguments."
else
  echo "First command-line argument is $1."
fi

echo

exit 0
```

在 Bash 里， `[[ ]]` 是比 `[ ]` 更加通用的写法。其作为扩展 `test` 命令从 ksh88 中被继承了过来。

在 `[[` 和 `]]` 中不会进行文件名扩展或字符串分割，但是可以进行参数扩展和命令替换。

```
file=/etc/passwd

if [[ -e $file ]]
then
  echo "Password file exists."
fi
```

使用 `[[...]]` 代替 `[...]` 可以避免很多逻辑错误。比如可以在 `[[]]` 中使用 `&&` ，`||` ，`<` 和 `>` 操作符，而在 `[]` 中使用则会报错。

在 `[[]]` 中会自动执行八进制和十六进制的进制转换操作。

```
# [[ 八进制和十六进制进制转换 ]]
# 感谢 Moritz Gronbach 提出。


decimal=15
octal=017    # = 15 (十进制)
hex=0x0f     # = 15 (十进制)

if [ "$decimal" -eq "$octal" ]
then
  echo "$decimal equals $octal"
else
  echo "$decimal is not equal to $octal"       # 15 不等于 017
fi      # 在单括号 [ ] 之间不会进行进制转换。


if [[ "$decimal" -eq "$octal" ]]
then
  echo "$decimal equals $octal"                # 15 等于 017
else
  echo "$decimal is not equal to $octal"
fi      # 在双括号 [[ ]] 之间会进行进制转换。

if [[ "$decimal" -eq "$hex" ]]
then
  echo "$decimal equals $hex"                  # 15 等于 0x0f
else
  echo "$decimal is not equal to $hex"
fi      # 十六进制也可以进行转换。
```

语法上并不严格要求在 `if` 之后一定要写 `test` 命令或者测试结构（`[]` 或 `[[]]`）。

```
dir=/home/bozo

if cd "$dir" 2>/dev/null; then   # "2>/dev/null" 重定向消去错误输出。

  echo "Now in $dir."
else
  echo "Can't change to $dir."
fi
```

`if COMMAND` 的退出状态就是 `COMMAND` 的退出状态。

同样的，测试括号也不一定需要与 `if` 一起使用。其可以同 列表结构 结合而不需要 `if` 。

```
var1=20
var2=22
[ "$var1" -ne "$var2" ] && echo "$var1 is not equal to $var2"

home=/home/bozo
[ -d "$home" ] || echo "$home directory does not exist."
```

`(( ))` 结构 扩展和执行算术表达式。如果执行结果为0，其返回的 退出状态码 为1（假）。非0表达式返回的退出状态为0（真）。这与上述所使用的 `test` 和 `[ ]` 结构形成鲜明的对比。

样例 7-3. 使用 `(( ))` 进行算术测试

```
#!/bin/bash
# arith-tests.sh
# 算术测试。

# (( ... )) 结构执行并测试算术表达式。
# 与 [ ... ] 结构的退出状态正好相反。

(( 0 ))
echo "Exit status of \"(( 0 ))\" is $?."         # 1

(( 1 ))
```

```
echo "Exit status of \"(( 1 ))\" is $?."         # 0

(( 5 > 4 ))                                      # 真
echo "Exit status of \"(( 5 > 4 ))\" is $?."     # 0

(( 5 > 9 ))                                      # 假
echo "Exit status of \"(( 5 > 9 ))\" is $?."     # 1

(( 5 == 5 ))                                     # 真
echo "Exit status of \"(( 5 == 5 ))\" is $?."    # 0
# (( 5 = 5 )) 会报错。

(( 5 - 5 ))                                      # 0
echo "Exit status of \"(( 5 - 5 ))\" is $?."     # 1

(( 5 / 4 ))                                      # 合法
echo "Exit status of \"(( 5 / 4 ))\" is $?."     # 0

(( 1 / 2 ))                                      # 结果小于1
echo "Exit status of \"(( 1 / 2 ))\" is $?."     # 舍入至0。
                                                 # 1

(( 1 / 0 )) 2>/dev/null                          # 除0，非法
#           ^^^^^^^^^^^
echo "Exit status of \"(( 1 / 0 ))\" is $?."     # 1

# "2>/dev/null" 的作用是什么？
# 如果将其移除会发生什么？
# 尝试移除这条语句并重新执行脚本。

# ======================================= #

# (( ... )) 在 if-then 中也非常有用

var1=5
var2=4

if (( var1 > var2 ))
then #^      ^      注意不是 $var1 和 $var2，为什么？
  echo "$var1 is greater then $var2"
```

```
fi      # 5 大于 4

exit 0
```

> 1. 标记是一个具有特殊意义（元语义）的符号或者短字符串。在 Bash 里像 `[` 和 `.（点命令）` 这样的标记可以扩展成关键字和命令。↩

7.2 文件测试操作

下列每一个 `test` 选项在满足条件时，返回0（真）。

-e

检测文件是否存在

-a

检测文件是否存在

等价于 `-e` 。不推荐使用，已被弃用[1]。

-f

文件是常规文件(regular file)，而非目录或 设备文件

-s

文件大小不为0

-d

文件是一个目录

-b

文件是一个 块设备

-c

文件是一个 字符设备

```
device0="/dev/sda2"    # /   (根目录)
if [ -b "$device0" ]
then
  echo "$device0 is a block device."
fi

# /dev/sda2 是一个块设备。



device1="/dev/ttyS1"   # PCMCIA 调制解调卡
if [ -c "$device1" ]
then
  echo "$device1 is a character device."
fi

# /dev/ttyS1 是一个字符设备。
```

-p

文件是一个 管道设备

```
function show_input_type()
{
   [ -p /dev/fd/0 ] && echo PIPE || echo STDIN
}

show_input_type "Input"                          # STDIN
echo "Input" | show_input_type                   # PIPE

# 这个例子由 Carl Anderson 提供。
```

-h

文件是一个 符号链接

-L

文件是一个符号链接

-S

文件是一个 套接字

-t

文件（文件描述符）与终端设备关联

该选项通常被用于 测试 脚本中的 `stdin [ -t 0 ]` 或 `stdout [ -t 1 ]` 是否为终端设备。

-r

该文件对执行测试的用户可读

-w

该文件对执行测试的用户可写

-x

该文件可被执行测试的用户所执行

-g

文件或目录设置了 set-group-id `sgid` 标志

如果一个目录设置了 `sgid` 标志，那么在该目录中所有的新建文件的权限组都归属于该目录的权限组，而非文件创建者的权限组。该标志对共享文件夹很有用。

-u

文件设置了 set-user-id `suid` 标志。

一个属于 root 的可执行文件设置了 `suid` 标志后，即使是一个普通用户执行也拥有 root 权限[2]。对需要访问硬件设备的可执行文件（例如 `pppd` 和 `cdrecord` ）很有用。如果没有 `suid` 标志，这些可执行文件就不能被非 root 用户所调用了。

```
-rwsr-xr-t    1 root       178236 Oct  2  2000 /usr/sbin/pppd
```

设置了 `suid` 标志后，在权限中会显示 `s` 。

-k

设置了粘滞位(sticky bit)。

标志粘滞位是一种特殊的文件权限。如果文件设置了粘滞位，那么该文件将会被存储在高速缓存中以便快速访问[3]。如果目录设置了该标记，那么它将会对目录的写权限进行限制，目录中只有文件的拥有者可以修改或删除文件。设置标记后你可以在权限中看到 `t` 。

```
drwxrwxrwt    7 root         1024 May 19 21:26 tmp/
```

如果一个用户不是设置了粘滞位目录的拥有者，但对该目录有写权限，那么他仅仅可以删除目录中他所拥有的文件。这可以防止用户不经意间删除或修改其他人的文件，例如 `/tmp` 文件夹。（当然目录的所有者可以删除或修改该目录下的所有文件）

-O

执行用户是文件的拥有者

-G

文件的组与执行用户的组相同

-N

文件在在上次访问后被修改过了

f1 -nt f2

文件 f1 比文件 f2 新

f1 -ot f2

文件 f1 比文件 f2 旧

f1 -ef f2

文件 f1 和文件 f2 硬链接到同一个文件

!

取反——对测试结果取反(如果条件缺失则返回真)。

样例 7-4. 检测链接是否损坏

```
#!/bin/bash
# broken-link.sh
# Lee bigelow <ligelowbee@yahoo.com> 编写。
# ABS Guide 经许可可以使用。

#  该脚本用来发现输出损坏的链接。输出的结果是被引用的，
#+ 所以可以直接导到 xargs 中进行处理 :)
#  例如：sh broken-link.sh /somedir /someotherdir|xargs rm
#
#  更加优雅的方式：
#
#  find "somedir" -type l -print0|\
#  xargs -r0 file|\
#  grep "broken symbolic"|
#  sed -e 's/^\|: *broken symbolic.*$/"/g'
#
#  但是这种方法不是纯 Bash 写法。
#  警告：小心 /proc 文件下的文件和任意循环链接！
```

```
##############################################


#  如果不给脚本传任何参数，那么 directories-to-search 设置为当前目录
#+ 否则设置为传进的参数
######################

[ $# -eq 0 ] && directory=`pwd` || directory=$@


#  函数 linkchk 是用来检测传入的文件夹中是否包含损坏的链接文件，
#+ 并引用输出他们。
#  如果文件夹中包含子文件夹，那么将子文件夹继续传给 linkchk 函数进行检测。
##################

linkchk () {
    for element in $1/*; do
      [ -h "$element" -a ! -e "$element" ] && echo \"$element\"
      [ -d "$element" ] && linkchk $element
    # -h 用来检测是否是链接，-d 用来检测是否是文件夹。
    done
}

#  检测传递给 linkchk() 函数的参数是否是一个存在的文件夹，
#+ 如果不是则报错。
################
for directory in $direcotrys; do
    if [ -d $directory ]
        then linkchk $directory
        else
            echo "$directory is not a directory"
            echo "Usage $0 dir1 dir2 ..."
    fi
done

exit $?
```

样例 31-1，样例 11-8，样例 11-3，样例 31-3和样例 A-1 也包含了文件测试操作符的使用。

1

1. 摘自1913年版本的韦氏词典

```
Deprecate
...

To pray against, as an evil;
to seek to avert by prayer;
to desire the removal of;
to seek deliverance from;
to express deep regret for;
to disapprove of strongly.
```

↩

2. 注意使用 suid 的可执行文件可能会带来安全问题。suid 标记对 shell 脚本没有影响。↩

3. 在 Linux 系统中，文件已经不使用粘滞位了, 粘滞位只作用于目录。↩

7.3 其他比较操作

二元比较操作可以比较变量或者数量。 需要注意的是，整数和字符串比较使用的是两套不同的操作符。

整数比较

-eq

等于

```
if [ "$a" -eq "$b" ]
```

-ne

不等于

```
if [ "$a" -ne "$b" ]
```

-gt

大于

```
if [ "$a" -gt "$b" ]
```

-ge

大于等于

```
if [ "$a" -ge "$b" ]
```

-lt

小于

```
if [ "$a" -lt "$b" ]
```

-le

小于等于

```
if [ "$a" -le "$b" ]
```

<

小于（使用 双圆括号）

```
(("$a" < "$b"))
```

<=

小于等于（使用双圆括号）

```
(("$a" <= "$b"))
```

>

大于（使用双圆括号）

```
(("$a" > "$b"))
```

>=

大于等于（使用双圆括号）

```
(("$a" >= "$b"))
```

字符串比较

=

等于

```
if [ "$a" = "$b" ]
```

⚠️ 注意在 `=` 前后要加上空格

`if [ "$a"="$b" ]` 和上面不等价。

==

等于

`if [ "$a" == "$b" ]`

和 `=` 同义

 `==` 操作符在 双方括号 和单方括号里的功能是不同的。

```
[[ $a == z* ]]    # $a 以 "z" 开头时为真（模式匹配）
[[ $a == "z*" ]] # $a 等于 z* 时为真（字符匹配）

[ $a == z* ]      # 发生文件匹配和字符分割。
[ "$a" == "z*" ] # $a 等于 z* 时为真（字符匹配）

# 感谢 Stéphane Chazelas
```

!=

不等于

`if [ "$a" != "$b" ]`

在 `[[ ... ]]` 结构中会进行模式匹配。

<

小于，按照 ASCII码 排序。

`if [[ "$a" < "$b" ]]`

`if [ "$a" \< "$b" ]`

注意在 `[]` 结构里 `<` 需要被 转义。

>

大于，按照 ASCII 码排序。

`if [[ "$a" > "$b" ]]`

`if [ "$a" \> "$b" ]`

注意在 `[]` 结构里 `>` 需要被转义。

样例 27-11 包含了比较操作符。

-z

字符串为空，即字符串长度为0。

```
String=''   # 长度为0的字符串变量。

if [ -z "$String" ]
then
  echo "\$String is null."
else
  echo "\$String is NOT null."
fi     # $String is null.
```

-n

字符串非空（ `null` ）。

⚠ 使用 `-n` 时字符串必须是在括号中且被引用的。使用 `! -z` 判断未引用的字符串或者直接判断（样例 7-6）通常可行，但是非常危险。判断字符串时一定要引用[1]。

样例 7-5. 算术比较和字符串比较

```
#!/bin/bash

a=4
b=5

# 这里的 "a" 和 "b" 可以是整数也可以是字符串。
# 因为 Bash 的变量是弱类型的，因此字符串和整数比较有很多相同之处。

# 在 Bash 中可以用处理整数的方式来处理全是数字的字符串。
# 但是谨慎使用。

echo

if [ "$a" -ne "$b" ]
then
  echo "$a is not equal to $b"
  echo "(arithmetic comparison)"
fi

echo

if [ "$a" != "$b" ]
then
  echo "$a is not equal to $b."
  echo "(string comparison)"
  #     "4"  != "5"
  # ASCII 52 != ASCIII 53
fi

# 在这个例子里 "-ne" 和 "!=" 都可以。

echo

exit 0
```

样例 7-6. 测试字符串是否为空（ `null` ）

```
#!/bin/bash
# str-test.sh: 测试是否为空字符串或是未引用的字符串。
```

```
# 使用 if [ ... ] 结构

# 如果字符串未被初始化，则其值是未定义的。
# 这种状态就是空 "null"（并不是 0）。

if [ -n $string1 ]    # 并未声明或是初始化 string1。
then
  echo "String \"string1\" is not null."
else
  echo "String \"string1\" is null."
fi
# 尽管没有初始化 string1，但是结果显示其非空。

echo

# 再试一次。

if [ -n "$string1" ]   # 这次引用了 $string1。
then
  echo "String \"string1\" is not null."
else
  echo "String \"string1\" is null."
fi                    # 在测试括号内引用字符串得到了正确的结果。

echo

if [ $string1 ]       # 这次只有一个 $string1。
then
  echo "String \"string1\" is not null."
else
  echo "String \"string1\" is null."
fi                    # 结果正确。
# 独立的 [ ... ] 测试操作符可以用来检测字符串是否为空。
# 最好将字符串进行引用（if [ "$string1" ]）。
#
# Stephane Chazelas 指出：
#    if [ $string1 ]    只有一个参数 "]"
#    if [ "$string1" ]  则有两个参数，空的 "$string1" 和 "]"
```

```
echo


string1=initialized

if [ $string1 ]         # $string1 这次仍然没有被引用。
then
  echo "String \"string1\" is not null."
else
  echo "String \"string1\" is null."
fi                      # 这次的结果仍然是正确的。
# 最好将字符串引用（"$string1"）


string1="a = b"

if [ $string1 ]         # $string1 这次仍然没有被引用。
then
  echo "String \"string1\" is not null."
else
  echo "String \"string1\" is null."
fi                      # 这次没有引用就错了。

exit 0   # 同时感谢 Florian Wisser 的提示。
```

样例 7-7. `zmore`

```
#!/bin/bash
# zmore

# 使用筛选器 'more' 查看 gzipped 文件。

E_NOARGS=85
E_NOTFOUND=86
E_NOTGZIP=87

if [ $# -eq 0 ] # 作用和 if [ -z "$1" ] 相同。
# $1 可以为空： zmore "" arg2 arg3
```

```
then
  echo "Usage: `basename $0` filename" >&2
  # 将错误信息通过标准错误 stderr 进行输出。
  exit $E_NOARGS
  # 脚本的退出状态为 85.
fi

filename=$1

if [ ! -f "$filename" ]   # 引用字符串以防字符串中带有空格。
then
  echo "File $filename not found!" >&2   # 通过标准错误 stderr 进行
输出。
  exit $E_NOTFOUND
fi

if [ ${filename##*.} != "gz" ]
# 在括号内使用变量代换。
then
  echo "File $1 is not a gzipped file!"
  exit $E_NOTGZIP
fi

zcat $1 | more

# 使用筛选器 'more'
# 也可以用 'less' 替代

exit $?   # 脚本的退出状态由管道 pipe 的退出状态决定。
#  实际上 "exit $?" 不一定要写出来，
#+ 因为无论如何脚本都会返回最后执行命令的退出状态。
```

复合比较

-a

逻辑与

`exp1 -a exp2` 返回真当且仅当 `exp1` 和 `exp2` 均为真。

-o

逻辑或

如果 `exp1` 或 `exp2` 为真，则 `exp1 -o exp2` 返回真。

以上两个操作和 双方括号 结构中的 Bash 比较操作符号 `&&` 和 `||` 类似。

```
[[ condition1 && condition2 ]]
```

测试操作 `-o` 和 `-a` 可以在 `test` 命令或在测试括号中进行。

```
if [ "$expr1" -a "$expr2" ]
then
  echo "Both expr1 and expr2 are true."
else
  echo "Either expr1 or expr2 is false."
fi
```

rihad 指出：

```
[ 1 -eq 1 ] && [ -n "`echo true 1>&2`" ]   # 真
[ 1 -eq 2 ] && [ -n "`echo true 1>&2`" ]   # 没有输出
# ^^^^^^^ 条件为假。到这里为止，一切都按预期执行。

# 但是
[ 1 -eq 2 -a -n "`echo true 1>&2`" ]       # 真
# ^^^^^^^ 条件为假。但是为什么结果为真？

# 是因为括号内的两个条件子句都执行了么？
[[ 1 -eq 2 && -n "`echo true 1>&2`" ]]     # 没有输出
# 并不是。

#  所以显然 && 和 || 具备“短路”机制，
#+ 例如对于 &&，若第一个表达式为假，则不执行第二个表达式直接返回假，
#+ 而 -a 和 -o 则不是。
```

复合比较操作的例子可以参考 样例 8-3，样例 27-17 和 样例 A-29。

> 1. S.C. 指出在复合测试中，仅仅引用字符串可能还不够。比如表达式 `[ -n "$string" -o "$a" = "$b" ]` 在某些 Bash 版本下，如果 `$string` 为空可能会出错。更加安全的方式是，对于可能为空的字符串，添加一个额外的字符，例如 `[ "x$string" != x -o "x$a" = "x$b" ]`（其中的 x 互相抵消）。↩

7.4 嵌套 if/then 条件测试

可以嵌套 `if/then` 条件测试结构。嵌套的结果等价于使用 `&&` 复合比较操作符。

```
a=3

if [ "$a" -gt 0 ]
then
  if [ "$a" -lt 5 ]
  then
    echo "The value of \"a\" lies somewhere between 0 and 5."
  fi
fi

# 和下面的结果相同

if [ "$a" -gt 0 ] && [ "$a" -lt 5 ]
then
  echo "The value of \"a\" lies somewhere between 0 and 5."
fi
```

在 样例 37-4 和 样例 17-11 中展示了嵌套 `if/then` 条件测试结构。

7.5 牛刀小试

系统文件 `xinitrc` 可以用来启动软件 X Server。该文件包含了许多 `if/then` 测试结构。下面的代码摘录自较早版本的 `xinitrc` （大约在 Red Hat 7.1 版本）。

```
if [ -f $HOME/.Xclients ]; then
  exec $HOME/.Xclients
elif [ -f /etc/X11/xinit/Xclients ]; then
  exec /etc/X11/xinit/Xclients
else
    # 安全分支。尽管程序不会执行这个分支。
    # （我们在 Xclients 中也提供了相同的机制）增强程序可靠性。
    xclock -geometry 100x100-5+5 &
    xterm -geometry 80x50-50+150 &
    if [ -f /usr/bin/netscape -a -f /usr/share/doc/HTML/index.ht
ml ]; then
            netscape /usr/share/doc/HTML/index.html
    fi
fi
```

试着解释代码片段中的条件测试结构, 然后试着在 /etc/X11/xinit/xinitrc 查看最新版本，并且分析其中的 if/then 条件测试结构。为了更好的进行分析，你可能需要继续阅读后面章节中对 `grep` ， `sed` 和 正则表达式 的讨论。

第八章 运算符相关话题

本章目录

- 8.1 运算符
- 8.2 数字常量
- 8.3 双圆括号结构
- 8.4 运算符优先级

8.1. 运算符

赋值运算符

变量赋值，初始化或改变一个变量的值。

=

等号 `=` 赋值运算符，既可用于算术赋值，也可用于字符串赋值。

```
var=27
category=minerals  # "="左右不允许有空格
```

>  注意，不要混淆 `=` 赋值运算符与 `=` 测试操作符。

```
#   =   作为测试操作符

if [ "$string1" = "$string2" ]
then
   command
fi

#  [ "X$string1" = "X$string2" ] 这样写是安全的,
#  这样写可以避免任意一个变量为空时的报错。
#  (变量前加的"X"字符规避了变量为空的情况)
```

算术运算符

+

加

-

减

*

乘

/

除

**

幂运算

```
# Bash, 2.02版本，推出了"**"幂运算操作符。

let "z=5**3"    # 5 * 5 * 5
echo "z = $z"   # z = 125
```

%

取余(返回整数除法的余数)

```
bash$ expr 5 % 3
2
```

5/3=1，余2 取余运算符经常被用于生成一定范围内的数(案例9-11, 案例9-15)，以及格式化程序输出(案例 27-16，案例 A-6)。 取余运算符还可以用来产生素数（案例A-15），取余的出现大大扩展了整数的算术运算。

样例 **8-1.** 最大公约数

```
#!/bin/bash
# gcd.sh: 最大公约数
#         使用欧几里得算法
```

```
#  两个整数的最大公约数（gcd）
#  是两数能同时整除的最大数

#  欧几里得算法使用辗转相除法
#    In each pass,
#       dividend <---  divisor
#       divisor  <---  remainder
#    until remainder = 0.
#    The gcd = dividend, on the final pass.
#
#  关于欧几里得算法更详细的讨论，可以查看：
#  Jim Loy's site, http://www.jimloy.com/number/euclids.htm.


# ------------------------------------------------------
# 参数检查
ARGS=2
E_BADARGS=85

if [ $# -ne "$ARGS" ]
then
  echo "Usage: `basename $0` first-number second-number"
  exit $E_BADARGS
fi
# ------------------------------------------------------


gcd ()
{

  dividend=$1             #  随意赋值，
  divisor=$2              #  两数谁大谁小是无关紧要的，
                          #  为什么?

  remainder=1             #  如果在测试括号里使用了一个未初始化的变量，
                          #  会报错的。

  until [ "$remainder" -eq 0 ]
  do    #  ^^^^^^^^^^  该变量必须在使用前初始化！
    let "remainder = $dividend % $divisor"
```

```
    dividend=$divisor      # 对被除数，除数重新赋值
    divisor=$remainder
  done                     # 欧几里得算法

}                          # 最后的 $dividend 就是最大公约数（gcd）


gcd $1 $2

echo; echo "GCD of $1 and $2 = $dividend"; echo


# 练习 :
# ---------
# 1) 检查命令行参数，保证其为整数，
#+   如果有错误，捕捉错误并在脚本退出前打印出适当的错误信息。
# 2) 使用本地变量(local variables)重写gcd()函数。

exit 0
```

+=

加等 （加上一个数）[1] `let "var += 5"` 的结果是 `var` 变量的值增加了5。

-=

减等 （减去一个数）

*=

乘等 （乘以一个数） `let "var *= 4"` 的结果是 `var` 变量的值乘了4。

/=

除等 （除以一个数）

%=

余等（取余赋值）

小结

算术运算符常用于 `expr` 或 `let` 表达式中。

样例 **8-2.** 使用算术运算符

```
#!/bin/bash
# 使变量自增1，10种不同的方法实现

n=1; echo -n "$n "

let "n = $n + 1"   # 可以使用 let "n = n + 1"
echo -n "$n "



: $((n = $n + 1))
#  ":" 是必要的，不加的话，bash会将
#+ "$((n = $n + 1))"看做一条命令。
echo -n "$n "

(( n = n + 1 ))
#  更简洁的写法。
#  感谢 David Lombard指出。
echo -n "$n "

n=$(($n + 1))
echo -n "$n "

: $[ n = $n + 1 ]
#  ":" 是必要的，不加的话，bash会将
#+ "$[ n = $n + 1 ]"看做一条命令。
#  即使"n"是字符串，也是可行的。
echo -n "$n "

n=$[ $n + 1 ]
#  即使"n"是字符串，也是可行的。
#* 不要用这种写法，它已被废弃且不具有兼容性。
#  感谢 Stephane Chazelas.
```

```
echo -n "$n "

# 使用C风格的自增运算符也是可以的
# 感谢 Frank Wang 指出。

let "n++"          # let "++n" 可行
echo -n "$n "

(( n++ ))          # (( ++n ))  可行
echo -n "$n "

: $(( n++ ))       # : $(( ++n )) 可行
echo -n "$n "

: $[ n++ ]         # : $[ ++n ] 可行
echo -n "$n "

echo

exit 0
```

在早期的Bash版本中，整型变量是带符号的长整型数（32-bit），取值范围从 -2147483648 到 2147483647。如果算术操作超出了整数的取值范围，结果会不准确。

```
echo $BASH_VERSION   # Bash 1.14版本

a=2147483646
echo "a = $a"        # a = 2147483646
let "a+=1"           # 自增 "a".
echo "a = $a"        # a = 2147483647
let "a+=1"           # 再次自增"a"，超出取值范围。
echo "a = $a"        # a = -2147483648
                     #      错误：超出范围，
                     #+     最左边的符号位被重置，
                     #+     结果变负
```

Bash版本 >= 2.05b, Bash支持了64-bit整型数。

>  注意，Bash并不支持浮点运算，Bash会将带小数点的数看做字符串。

```
a=1.5

let "b = $a + 1.3"  # 报错
# t2.sh: let: b = 1.5 + 1.3: syntax error in expression
#                            (error token is ".5 + 1.3")

echo "b = $b"       # b=1
```

如果你想在脚本中使用浮点数运算，借助bc或外部数学函数库吧。

位运算

位运算很少出现在shell脚本中，在bash中加入位运算的初衷似乎是为了操控和检测来自 `ports` 或 `sockets` 的数据。位运算在编译型语言中能发挥更大的作用，比如C/C++，位运算提供了直接访问系统硬件的能力。然而，聪明的vladz在他的base64.sh(案例 A-54)脚本中也用到了位运算。 下面介绍位运算符。

<<

左移运算符(左移1位相当于乘2)

<<=

左移赋值

`let "var <<= 2"` 的结果是var变量的值向左移了2位(乘以4)

>>

右移运算符(右移1位相当于除2)

>>=

右移赋值

&

按位与（AND）

&=

按位与等（AND-equal）

|

按位或（OR）

|=

按位或等（OR-equal）

~

按位取反

^

按位异或（XOR）

^=

按位异或等（XOR-equal）

逻辑(布尔)运算符

!

非(NOT)

```
if [ ! -f $FILENAME ]
then
  ...
```

&&

与(AND)

```
if [ $condition1 ] && [ $condition2 ]
#  等同于:  if [ $condition1 -a $condition2 ]
#  返回true如果 condition1 和 condition2 同时为真...

if [[ $condition1 && $condition2 ]]    # 可行
#  注意，&& 运算符不能用在[ ... ]结构里。
```

> &&也可以被用在 `list` 结构中连接命令。

||

或(OR)

```
if [ $condition1 ] || [ $condition2 ]

#  等同于:  if [ $condition1 -a $condition2 ]
#  返回true如果 condition1 和 condition2 任意一个为真...

if [[ $condition1 || $condition2 ]]    # 可行
#  注意，|| 运算符不能用在[ ... ]结构里。
```

小结

样例 **8-3.** 在条件测试中使用 **&&** 和 **||**

```
#!/bin/bash
```

```
a=24
b=47

if [ "$a" -eq 24 ] && [ "$b" -eq 47 ]
then
  echo "Test #1 succeeds."
else
  echo "Test #1 fails."
fi

#  错误:   if [ "$a" -eq 24 && "$b" -eq 47 ]
#           这样写的话，bash会先执行'[ "$a" -eq 24'
#           然后就找不到右括号']'了...
#
#  注意:  if [[ $a -eq 24 && $b -eq 24 ]]  这样写是可以的
#  双方括号测试结构比单方括号更加灵活。
#  (双方括号中的"&&"与单方括号中的"&&"意义不同)
#  感谢 Stephane Chazelas 指出。


if [ "$a" -eq 98 ] || [ "$b" -eq 47 ]
then
  echo "Test #2 succeeds."
else
  echo "Test #2 fails."
fi



#  使用 -a 和 -o 选项也具有同样的效果。
#  感谢 Patrick Callahan 指出。


if [ "$a" -eq 24 -a "$b" -eq 47 ]
then
  echo "Test #3 succeeds."
else
  echo "Test #3 fails."
fi
```

```
if [ "$a" -eq 98 -o "$b" -eq 47 ]
then
  echo "Test #4 succeeds."
else
  echo "Test #4 fails."
fi



a=rhino
b=crocodile
if [ "$a" = rhino ] && [ "$b" = crocodile ]
then
  echo "Test #5 succeeds."
else
  echo "Test #5 fails."
fi

exit 0
```

`&&` 和 `||` 运算符也可以用在算术运算中。

```
bash$ echo $(( 1 && 2 )) $((3 && 0)) $((4 || 0)) $((0 || 0))
1 0 1 0
```

其他运算符

,

逗号运算符 逗号运算符用于连接两个或多个算术操作，所有的操作会被依次求值（可能会有副作用）。[2]

```
let "t1 = ((5 + 3, 7 - 1, 15 - 4))"
echo "t1 = $t1"           ^^^^^^  # t1 = 11
# 这里的t1 被赋值了11，为什么？

let "t2 = ((a = 9, 15 / 3))"      # 对"a"赋值并对"t2"求值。
echo "t2 = $t2    a = $a"         # t2 = 5    a = 9
```

逗号运算符常被用在 `for` 循环中。参看案例 11-13。

> 1. 取决与不同的上下文，+= 也可能作为字符串连接符。它可以很方便地修改环境变量。↩

> 2. 副作用，顾名思义，就是预料之外的结果。↩

8.2. 数字常量

通常情况下，shell脚本会把数字以十进制整数看待(base 10)，除非数字加了特殊的前缀或标记。 带前缀0的数字是八进制数(base 8)；带前缀0x的数字是十六进制数(base 16)。 内嵌 # 的数字会以 BASE#NUMBER 的方式进行求值（不能超出当前shell支持整数的范围）。

样例 **8-4.** 数字常量的表示

```
#!/bin/bash
# numbers.sh: 不同进制数的表示

# 十进制数: 默认
let "dec = 32"
echo "decimal number = $dec"             # 32
# 一切正常。


# 八进制数: 带前导'0'的数
let "oct = 032"
echo "octal number = $oct"               # 26
# 结果以 十进制 打印输出了。
# ------ ------ -----------


# 十六进制数: 带前导'0x'或'0X'的数
let "hex = 0x32"
echo "hexadecimal number = $hex"         # 50

echo $((0x9abc))                         # 39612
#     ^^      ^^   双圆括号进行表达式求值
# 结果以十进制打印输出。



# 其他进制数: BASE#NUMBER
# BASE 范围:  2 - 64
# NUMBER 必须以 BASE 规定的正确形式书写，如下:
```

```
let "bin = 2#111100111001101"
echo "binary number = $bin"              # 31181

let "b32 = 32#77"
echo "base-32 number = $b32"             # 231

let "b64 = 64#@_"
echo "base-64 number = $b64"             # 4031

# 这种表示法只对进制范围(2 - 64)内的 ASCII 字符有效。
# 10 数字 + 26 小写字母 + 26 大写字母 + @ + _


echo

echo $((36#zz)) $((2#10101010)) $((16#AF16)) $((53#1aA))
                                         # 1295 170 44822 3375

#  重要提醒:
#  ---------
#  使用超出进制范围以外的符号会报错。

let "bad_oct = 081"

# (可能的) 报错信息:
#  bad_oct = 081: value too great for base (error token is "081"
)
#              Octal numbers use only digits in the range 0 - 7.

exit $?        # 退出码 = 1 (错误)

# 感谢 Rich Bartell 和 Stephane Chazelas 的说明。
```

双圆括号结构

与 `let` 命令类似，`(( ... ))` 结构允许对算术表达式的扩展和求值。它是 `let` 命令的简化形式。例如，a=$((5 + 3)) 会将变量a赋值成 5 + 3，也就是8。在Bash中，双圆括号结构也允许以C风格的方式操作变量。例如，((var++))。

样例 **8-5.** 以**C**风格的方式操作变量

```
#!/bin/bash
# c-vars.sh
# 以C风格的方式操作变量，使用(( ... ))结构


echo

(( a = 23 ))  #  C风格的变量赋值，注意"="等号前后都有空格

echo "a (initial value) = $a"   # 23

(( a++ ))     #  后缀自增'a'，C-style.
echo "a (after a++) = $a"       # 24

(( a-- ))     #  后缀自减'a', C-style.
echo "a (after a--) = $a"       # 23


(( ++a ))     #  前缀自增'a', C-style.
echo "a (after ++a) = $a"       # 24

(( --a ))     #  前缀自减'a', C-style.
echo "a (after --a) = $a"       # 23

echo

########################################################
#  注意，C风格的++，--运算符，前缀形式与后缀形式有不同的
#+ 副作用。
```

```
n=1; let --n && echo "True" || echo "False"  # False
n=1; let n-- && echo "True" || echo "False"  # True

#  感谢 Jeroen Domburg。
########################################################

echo

(( t = a<45?7:11 ))   # C风格三目运算符。
#       ^  ^ ^
echo "If a < 45, then t = 7, else t = 11."  # a = 23
echo "t = $t "                              # t = 7

echo



# -----------
# 复活节彩蛋!
# -----------
#  Chet Ramey 偷偷往Bash里加入了C风格的语句结构,
#  还没写文档说明 (实际上很多是从ksh中继承过来的)。
#  在Bash 文档中,Ramey把 (( ... ))结构称为shell 算术运算,
#  但是这种表述并不准确...
#  抱歉啊,Chet,把你的秘密抖出来了。

#  参看 "for" 和 "while" 循环章节关于 (( ... )) 结构的部分。

#  (( ... )) 结构在Bash 2.04版本之后才能正常工作。

exit
```

还可以参看 样例 **11-13** 与 样例 **8-4**。

运算符优先级

在脚本中，运算执行的顺序被称为优先级: 高优先级的操作会比低优先级的操作先执行。[1]

表 **8-1.** 运算符优先级**(**从高到低**)**

运算符	含义	注解	
var++ var--	后缀自增/自减	C风格运算符	
++var --var	前缀自增/自减		
! ~	按位取反/逻辑取反	对每一比特位取反/对逻辑判断的结果取反	
**	幂运算	算数运算符	
* / %	乘, 除, 取余	算数运算符	
+ -	加, 减	算数运算符	
<< >>	左移, 右移	比特位运算符	
-z -n	一元比较	字符串是/否为空	
-e -f -t -x, etc	一元比较	文件测试	
-lt -gt -le -ge <= >=	复合比较	字符串/整数比较	
-nt -ot -ef	复合比较	文件测试	
&	AND(按位与)	按位与操作	
^	XOR(按位异或)	按位异或操作	
\		OR(按位或)	按位或操作

&& -a	AND(逻辑与)	逻辑与，复合比较		
\	\	-o	OR(逻辑或)	逻辑或，复合比较
? :	if/else三目运算符	C风格运算符		
=	赋值	不要与test中的等号混淆		
*= /= %= += -= <<= >>= &=	赋值运算	先运算后赋值		
,	逗号运算符	连接一系列语句		

实际上，你只需要记住以下规则就可以了:

- 先乘除取余，后加减，与算数运算相似
- 复合逻辑运算符，&&, ||, -a, -o 优先级较低
- 优先级相同的操作按从左至右顺序求值

现在，让我们利用运算符优先级的知识来分析一下*Fedora Core Linux*中的 `/etc/init.d/functions` 文件。

```
while [ -n "$remaining" -a "$retry" -gt 0 ]; do

# 初看之下很恐怖...



# 分开来分析
while [ -n "$remaining" -a "$retry" -gt 0 ]; do
#       --condition 1-- ^^ --condition 2-

#  如果变量"$remaining" 长度不为0
#+      并且AND (-a)
#+ 变量 "$retry" 大于0
#+ 那么
#+ [ 方括号表达式 ] 返回成功(0)
#+ while-loop 开始迭代执行语句。
#  ==============================================================
```

```
=
#  "condition 1" 和 "condition 2" 在 AND之前执行，为什么？
#  因为AND(-a)优先级比-n,-gt来得低，逻辑与会在最后求值。
#################################################################
#

if [ -f /etc/sysconfig/i18n -a -z "${NOLOCALE:-}" ] ; then


# 同样，分开来分析
if [ -f /etc/sysconfig/i18n -a -z "${NOLOCALE:-}" ] ; then
#    --condition 1--------- ^^ --condition 2-----

#  如果文件"/etc/sysconfig/i18n" 存在
#+      并且AND (-a)
#+ 变量 $NOLOCALE 长度不为0
#+ 那么
#+ [ 方括号表达式 ] 返回成功(0)
#+ 执行接下来的语句。
#
#  和之前的情况一样，逻辑与AND(-a)最后求值。
#  因为在方括号测试结构中，逻辑运算的优先级是最低的。
#  ==============================================================
=
#  注意:
#  ${NOLOCALE:-} 是一个参数扩展式，看起来有点多余。
#  但是, 如果 $NOLOCALE 没有提前声明, 它会被设成null，
#  在某些情况下，这会有点问题。
```

> ⓘ为了避免在复杂比较运算中的错误，可以把运算分散到几个括号结构中。
>
> ```
> if [ "$v1" -gt "$v2"  -o  "$v1" -lt "$v2"  -a  -e "$filename
> " ]
> # 这样写不清晰...
>
> if [[ "$v1" -gt "$v2" ]] || [[ "$v1" -lt "$v2" ]] && [[ -e "
> $filename" ]]
> # 好多了 -- 把逻辑判断分散到多个组之中
> ```

1

> 1. Precedence(优先级)，根据上下文，与priority含义相近。 ↩

第三部分 shell进阶

目录

- 9. 换个角度看变量
  - 9.1 内部变量
  - 9.2 指定变量属性：`decalre` 或 `typeset`
  - 9.3 `$RANDOM`：随机产生整数
- 10. 变量处理
  - 10.1 字符串处理
    - 10.1.1 使用 `awk` 处理字符串
    - 10.1.2 参考资料
  - 10.2 参数替换
- 11. 循环与分支
  - 11.1 循环
  - 11.2 嵌套循环
  - 11.3 循环控制
  - 11.4 测试与分支
- 12. 命令替换
- 13. 算术扩展
- 14. 休息时间

第十章 变量处理

本章目录

- 10.1 字符串处理
  - 10.1.1 使用 `awk` 处理字符串
  - 10.1.2 参考资料
- 10.2 参数替换

10.1 字符串处理

Bash 支持的字符串操作数量达到了一个惊人的数目。但可惜的是，这些操作工具缺乏一个统一的核心。他们中的一些是参数代换的子集，另外一些则是 UNIX 下 `expr` 函数的子集。这将会导致语法前后不一致或者功能上出现重叠，更不用说那些可能导致的混乱了。

字符串长度

`$`

`expr length $string`

上面两个表达式等价于C语言中的 `strlen()` 函数。

`expr "$string" : '.*'`

```
stringZ=abcABC123ABCabc

echo ${#stringZ}                 # 15
echo `expr length $stringz`      # 15
echo `expr "$stringZ" : '.*'`    # 15
```

样例 10-1. 在文本的段落之间插入空行

```
#!/bin/bash
# paragraph-space.sh
# 版本 2.1，发布日期 2012年7月29日

# 在无空行的文本文件的段落之间插入空行。
# 像这样使用: $0 <FILENAME

MINLEN=60        # 可以试试修改这个值。它用来做判断。
#  假设一行的字符数小于 $MINLEN，并且以句点结束段落。
#+ 结尾部分有练习！

while read line  # 当文件有许多行的时候
do
  echo "$line"   # 输出行本身。

  len=${#line}
  if [[ "$len" -lt "$MINLEN" && "$line" =~ [*{\.}]$ ]]
# if [[ "$len" -lt "$MINLEN" && "$line" =~ \[*\.\] ]]
# 新版Bash将不能正常运行前一个版本的脚本。Ouch！
# 感谢 Halim Srama 指出这点，并且给出了修正版本。
    then echo    #  在该行以句点结束时，
  fi             #+ 增加一行空行。
done

exit

# 练习：
# -----
#  1) 该脚本通常会在文件的最后插入一个空行。
#+    尝试解决这个问题。
#  2) 在第17行仅仅考虑到了以句点作为句子终止的情况。
#+    修改以满足其他的终止符，例如 ?, ! 和 "。
```

起始部分字符串匹配长度

`expr match "$string" '$substring'`

其中，`$substring` 是一个正则表达式。

`expr "$string" : '$substring'`

其中，`$substring` 是一个正则表达式。

```
stringZ=abcABC123ABCabc
#       |------|
#       12345678

echo `expr match "$stringZ" 'abc[A-Z]*.2'`   # 8
echo `expr "$stringZ" : 'abc[A-Z]*.2'`       # 8
```

索引

`expr index $string $substring`

返回在 `$string` 中第一个出现的 `$substring` 字符所在的位置。

```
stringZ=abcABC123ABCabc
#       123456 ...
echo `expr index "$stringZ" C12`             # 6
                                             # C 的位置。

echo `expr index "$stringZ" 1c`              # 3
# 'c' (第三号位) 较 '1' 出现的更早。
```

几乎等价于C语言中的 `strchr()` 。

截取字符串（字符串分片）

`${string:position}`

在 `$string` 中截取自 `$position` 起的字符串。

如果参数 `$string` 是 "*" 或者 "@"，那么将会截取自 `$position` 起的位置参数。[1]

`${string:position:length}`

在 `$string` 中截取自 `$position` 起，长度为 `$length` 的字符串。

```
stringZ=abcABC123ABCabc
#       0123456789.....
#       索引位置从0开始。

echo ${stringZ:0}                            # abcABC123ABCabc
echo ${stringZ:1}                            # bcABC123ABCabc
echo ${stringZ:7}                            # 23ABCabc

echo ${stringz:7:3}                          # 23A
                                             # 三个字符的子字符串。




# 从右至左进行截取可行么？

echo ${stringZ:-4}                           # abcABC123ABCabc
# ${parameter:-default} 将会得到整个字符串。
# 但是……

echo ${stringZ:(-4)}                         # Cabc
echo ${stringZ: -4}                          # Cabc
# 现在可以了。
# 括号或者增加空格都可以"转义"位置参数。

# 感谢 Dan Jacobson 指出这些。
```

其中，参数 `position` 与 `length` 可以传入一个变量而不一定需要传入常量。

样例 10-2. 产生一个8个字符的随机字符串

```
#!/bin/bash
# rand-string.sh
# 产生一个8个字符的随机字符串。

if [ -n "$1" ]  #  如果在命令行中已经传入了参数，
then            #+ 那么就以它作为起始字符串。
  str0="$1"
else            #  否则，就将脚本的进程标识符PID作为起始字符串。
  str0="$$"
fi

POS=2  # 从字符串的第二位开始。
LEN=8  # 截取八个字符。

str1=$( echo "$str0" | md5sum | md5sum )
#                     ^^^^^^   ^^^^^^
# 将字符串通过管道计算两次 md5 来进行两次混淆。

randstring="${str1:$POS:$LEN}"
#                  ^^^^ ^^^^
# 允许传入参数

echo "$randstring"

exit $?

# bozo$ ./rand-string.sh my-password
# 1bdd88c4

# 不过不建议将其作为一种能够抵抗黑客的生成密码的方法。
```

如果参数 `$string` 是 "*" 或者 "@"，那么将会截取自 `$position` 起，最大个数为 `$length` 的位置参数。

```
echo ${*:2}          # 输出第二个及之后的所有位置参数。
echo ${@:2}          # 同上。

echo ${*:2:3}        # 从第二个位置参数起，输出三个位置参数。
```

`expr substr $string $position $length`

在 `$string` 中截取自 `$position` 起，长度为 `$length` 的字符串。

```
stringZ=abcABC123ABCabc
#       123456789......
#       索引位置从1开始。

echo `expr substr $stringZ 1 2`              # ab
echo `expr substr $stringZ 4 3`              # ABC
```

`expr match "$string" '\($substring\)'`

在 `$string` 中截取自 `$position` 起的字符串，其中 `$substring` 是正则表达式。

`expr "$string" : '\($substring\)'`

在 `$string` 中截取自 `$position` 起的字符串，其中 `$substring` 是正则表达式。

```
stringZ=abcABC123ABCabc
#       =======

echo `expr match "$stringZ" '\(.[b-c]*[A-Z]..[0-9]\)'`   # abcAB
C1
echo `expr "$stringZ" : '\(.[b-c]*[A-Z]..[0-9]\)'`       # abcAB
C1
echo `expr "$stringZ" : '\(.......\)'`                   # abcAB
C1
# 上面所有的形式都给出了相同的结果。
```

`expr match "$string" '.*\($substring\)'`

从 `$string` 结尾部分截取 `$substring` 字符串，其中 `$substring` 是正则表达式。

`expr "$string" : '.*\($substring\)'`

从 `$string` 结尾部分截取 `$substring` 字符串，其中 `$substring` 是正则表达式。

```
stringZ=abcABC123ABCabc
#                ======

echo `expr match "$stringZ" '.*\([A-C][A-C][A-C][a-c]*\)'`    # ABCabc
echo `expr "$stringZ" : '.*\(......\)'`                       # ABCabc
```

删除子串

`${string#substring}`

删除从 `$string` 起始部分起，匹配到的最短的 `$substring` 。

`${string##substring}`

删除从 `$string` 起始部分起，匹配到的最长的 `$substring` 。

```
stringZ=abcABC123ABCabc
#       |----|          最长
#       |----------|    最短

echo ${stringZ#a*C}      # 123ABCabc
# 删除 'a' 与 'c' 之间最短的匹配。

echo ${stringZ##a*C}     # abc
# 删除 'a' 与 'c' 之间最长的匹配。



# 你可以使用变量代替 substring。

X='a*C'

echo ${stringZ#$X}      # 123ABCabc
echo ${stringZ##$X}     # abc
                        # 同上。
```

`${string%substring}`

删除从 `$string` 结尾部分起，匹配到的最短的 `$substring` 。

例如：

```
# 将当前目录下所有后缀名为 "TXT" 的文件改为 "txt" 后缀。
# 例如 "file1.TXT" 改为 "file1.txt"。

SUFF=TXT
suff=txt

for i in $(ls *.$SUFF)
do
  mv -f $i $(i%.$SUFF).$suff
  #  除了从变量 $i 右侧匹配到的最短的字符串之外，
  #+ 其他一切都保持不变。
done ### 如果需要，循环可以压缩成一行的形式。

# 感谢 Rory Winston。
```

`${string%%substring}`

删除从 `$string` 结尾部分起，匹配到的最长的 `$substring` 。

```

stringZ=abcABC123ABCabc
#                    ||     最短
#        |------------|     最长

echo ${stringZ%b*c}      # abcABC123ABCa
# 从结尾处删除 'b' 与 'c' 之间最短的匹配。

echo ${stringZ%%b*c}     # a
# 从结尾处删除 'b' 与 'c' 之间最长的匹配。
```

这个操作对生成文件名非常有帮助。

样例 10-3. 改变图像文件的格式及文件名

```
#!/bin/bash
#  cvt.sh:
#  将目录下所有的 MacPaint 文件转换为 "pbm" 格式。
```

```
#  使用由 Brian Henderson (bryanh@giraffe-data.com) 维护的
#+ "netpbm" 包下的 "macptobpm" 二进制工具。
#  Netpbm 是大多数 Linux 发行版的标准组成部分。

OPERATION=macptopbm
SUFFIX=pbm          # 新的文件名后缀。

if [ -n "$1" ]
then
  directory=$1      # 如果已经通过脚本参数传入了目录名的情况……
else
  directory=$PWD    # 否则就使用当前工作目录。
fi

#  假设目标目录下的所有 MacPaint 图像文件都拥有
#+ ".mac" 的文件后缀名。

for file in $directory/*    # 文件名匹配。
do
  filename=${file%.*c}      #  从文件名中删除 ".mac" 后缀
                            #+ ('.*c' 匹配 '.' 与 'c' 之间的
                            #  所有字符，包括其本身)。
  $OPERATION $file > "$filename.$SUFFIX"
                            # 将转换结果重定向到新的文件。
  rm -f $file               # 在转换后删除原文件。
  echo "$filename.$SUFFIX"  # 将记录输出到 stdout 中。
done

exit 0

# 练习：
# -----
# 这个脚本会将当前工作目录下的所有文件进行转换。
# 修改脚本，使得它仅转换 ".mac" 后缀的文件。



# *** 还可以使用另外一种方法。 *** #

#!/bin/bash
```

```
# 将图像批处理转换成不同的格式。
# 假设已经安装了 imagemagick。（在大部分 Linux 发行版中都有）

INFMT=png   # 可以是 tif, jpg, gif 等等。
OUTFMT=pdf  # 可以是 tif, jpg, gif, pdf 等等。

for pic in *"$INFMT"
do
  p2=$(ls "$pic" | sed -e s/\.$INFMT//)
  # echo $p2
  convert "$pic" $p2.$OUTFMT
done

exit $?
```

样例 10-4. 将流音频格式转换成 ogg 格式

```
#!/bin/bash
# ra2ogg.sh: 将流音频文件 (*.ra) 转换成 ogg 格式。

# 使用 "mplayer" 媒体播放器程序：
#      http://www.mplayerhq.hu/homepage
# 使用 "ogg" 库与 "oggenc"：
#      http://www.xiph.org/
#
# 脚本同时需要安装一些解码器，例如 sipr.so 等等一些。
# 这些解码器可以在 compat-libstdc++ 包中找到。



OFILEPREF=${1%%ra}      # 删除 "ra" 后缀。
OFILESUFF=wav           # wav 文件后缀。
OUTFILE="$OFILEPREF""$OFILESUFF"
E_NOARGS=85

if [ -z "$1" ]          # 必须指定一个文件进行转换。
then
  echo "Usage: `basename $0` [filename]"
  exit $E_NOAGRS
fi
```

```
##########################################################
mplayer "$1" -ao pcm:file=$OUTFILE
oggenc "$OUTFILE"  # 由 oggenc 自动加上正确的文件后缀名。
##########################################################

rm "$OUTFILE"      # 立即删除 *.wav 文件。
                   # 如果你仍需保留原文件，注释掉上面这一行即可。

exit $?

#  注意：
#  -----
#  在网站上，点击一个 *.ram 的流媒体音频文件
#+ 通常只会下载到 *.ra 音频文件的 URL。
#  你可以使用 "wget" 或者类似的工具下载 *.ra 文件本身。


#  练习：
#  -----
#  这个脚本仅仅转换 *.ra 文件。
#  修改脚本增加适应性，使其可以转换 *.ram 或其他文件格式。
#
#  如果你非常有热情，你可以扩展这个脚本使其
#+ 可以自动下载并且转换流媒体音频文件。
#  给定一个 URL，自动下载流媒体音频文件 (使用 "wget")，
#+ 然后转换它。
```

下面是使用字符串截取结构对 `getopt` 的一个简单模拟。

样例 10-5. 模拟 `getopt`

```
#!/bin/bash
# getopt-simple.sh
# 作者: Chris Morgan
# 允许在高级脚本编程指南中使用。


getopt_simple()
```

```
{
    echo "getopt_simple()"
    echo "Parameters are '$*'"
    until [ -z "$1" ]
    do
      echo "Processing parameter of: '$1'"
      if [ ${1:0:1} = '/' ]
      then
          tmp=${1:1}               # 删除开头的 '/'
          parameter=${tmp%%=*}     # 取出名称。
          value=${tmp##*=}         # 取出值。
          echo "Parameter: '$parameter', value: '$value'"
          eval $parameter=$value
      fi
      shift
    done
}

# 将所有参数传递给 getopt_simple()。
getopt_simple $*

echo "test is '$test'"
echo "test2 is '$test2'"

exit 0  # 可以查看该脚本的修改版 UseGetOpt.sh。

---

sh getopt_example.sh /test=value1 /test2=value2

Parameters are '/test=value1 /test2=value2'
Processing parameter of: '/test=value1'
Parameter: 'test', value: 'value1'
Processing parameter of: '/test2=value2'
Parameter: 'test2', value: 'value2'
test is 'value1'
test2 is 'value2'
```

子串替换

`${string/substring/replacement}`

替换匹配到的第一个 `$substring` 为 `$replacement`。[2]

`${string//substring/replacement}`

替换匹配到的所有 `$substring` 为 `$replacement`。

```
stringZ=abcABC123ABCabc

echo ${stringZ/abc/xyz}        # xyzABC123ABCabc
                               # 将匹配到的第一个 'abc' 替换为 'xyz'。



echo ${stringZ//abc/xyz}       # xyzABC123ABCxyz
                               # 将匹配到的所有 'abc' 替换为 'xyz'。


echo  ---------------
echo "$stringZ"                # abcABC123ABCabc
echo  ---------------
                               # 字符串本身并不会被修改！

# 匹配以及替换的字符串可以是参数么？
match=abc
repl=000
echo ${stringZ/$match/$repl}  # 000ABC123ABCabc
#              ^      ^         ^^^
echo ${stringZ//$match/$repl} # 000ABC123ABC000
# Yes!          ^      ^        ^^^         ^^^

echo

# 如果没有给定 $replacement 字符串会怎样？
echo ${stringZ/abc}            # ABC123ABCabc
echo ${stringZ//abc}           # ABC123ABC
# 仅仅是将其删除而已。
```

`${string/#substring/replacement}`

替换 `$string` 中最前端匹配到的 `$substring` 为 `$replacement` 。

`${string/%substring/replacement}`

替换 `$string` 中最末端匹配到的 `$substring` 为 `$replacement` 。

```
stringZ=abcABC123ABCabc

echo ${stringZ/#abc/XYZ}          # XYZABC123ABCabc
                                  # 将前端的 'abc' 替换为 'XYZ'

echo ${stringZ/%abc/XYZ}          # abcABC123ABCXYZ
                                  # 将末端的 'abc' 替换为 'XYZ'
```

> 1. 这种情况同时适用于命令行参数和传入函数的参数。↩

> 2. 注意根据使用时上下文的不同，`$substring` 和 `$replacement` 可以是文本字符串也可以是变量。可以参考第一个样例。↩

10.1.1 使用 `awk` 处理字符串

在 Bash 脚本中可以调用字符串处理工具 `awk` 来替换内置的字符串处理操作。

样例 10-6. 使用另一种方式来截取和定位子字符串

```
#!/bin/bash
# substring-extraction.sh

String=23skidoo1
#      012345678    Bash
#      123456789    awk
# 注意不同字符串索引系统：
# Bash 中第一个字符的位置为0。
# Awk 中第一个字符的位置为1。

echo ${String:2:4} # 从第3位开始（0-1-2），4个字符的长度
                                         # skid

# Awk 中与 ${string:pos:length} 等价的是 substr(string,pos,length)。

echo | awk '
{ print substr("'"${String}"'",3,4)      # skid
}
'
#  将空的 "echo" 通过管道传递给 awk 作为一个模拟输入，
#+ 这样就不需要提供一个文件名来操作 awk 了。

echo "----"

# 同样的：

echo | awk '
{ print index("'"${String}"'", "skid")      # 3
}                                           # （skid 从第3位开始）
'   # 这里使用 awk 等价于 "expr index"。

exit 0
```

10.1.2 参考资料

更多关于于脚本中处理字符串的资料，可以查看 章节 10.2 以及 `expr` 命令的相关章节。

脚本样例：

1. 样例 16-9
2. 样例 10-9
3. 样例 10-10
4. 样例 10-11
5. 样例 10-13
6. 样例 A-36
7. 样例 A-41

10.2 参数替换

参数替换用来处理或扩展变量。

`${parameter}`

等同于 `$parameter`，是变量 parameter 的值。在一些特定的环境下，只允许使用不易混淆的 `${parameter}` 形式。

可以用于连接变量与字符串。

```
your_id=${USER}-on-${HOSTNAME}
echo "$your_id"
#
echo "Old \$PATH = $PATH"
PATH=${PATH}:/opt/bin  # 在脚本执行过程中临时在 $PATH 中加入 /opt/bin。

echo "New \$PATH = $PATH"
```

`${parameter-default}, ${parameter:-default}`

在没有设置变量的情况下使用缺省值。

```
var1=1
var2=2
# 没有设置 var3。

echo ${var1-$var2}   # 1
echo ${var3-$var2}   # 2
#           ^          注意前面的 $ 前缀。



echo ${username-`whoami`}
# 如果变量 $username 没有被设置，输出 `whoami` 的结果。
```

>  `${parameter-default}` 与 `${parameter:-default}` 的作用几乎相同，唯一不同的情况就是当变量 parameter 已经被声明但值为空时。

```
#!/bin/bash
# param-sub.sh

# 无论变量的值是否为空，其是否已被声明决定了缺省设置的触发。

username0=
echo "username0 has been declared, but is set to null."
echo "username0 = ${username0-`whoami`}"
# 将不会输出 `whoami` 的结果。

echo

echo username1 has not been declared.
echo "username1 = ${username1-`whoami`}"
# 将会输出 `whoami` 的结果。

username2=
echo "username2 has been declared, but is set to null."
echo "username2 = ${username2:-`whoami`}"
#                            ^
# 因为这里是 :- 而不是 -，所以将会输出 `whoami` 的结果。
# 与上面的 username0 比较。



#

# 再来一次：

variable=
# 变量已被声明，但其值为空。

echo "${varibale-0}"    # 没有输出。
echo "${variable:-1}"   # 1
#               ^

unser variable
```

```
echo "${variable-2}"    # 2
echo "${variable:-3}"   # 3

exit 0
```

当传入的命令行参数的数量不足时，可以使用这种缺省参数结构。

```
DEFAULT_FILENAME=generic.data
filename=${1:-$DEFAULT_FILENAME}
# 如果没有其他特殊情况，下面的代码块将会操作文件 "generic.data"。
# 代码块开始
# ...
# ...
# ...
# 代码块结束



# 摘自样例 "hanoi2.bash"：
DISKS=${1:-E_NOPARAM}   # 必须指定碟子的个数。
#  将 $DISKS 设置为传入的第一个命令行参数，
#+ 如果没有传入第一个参数，则设置为 $E_NOPARAM。
```

可以查看 样例 3-4，样例 31-2 和 样例 A-6。

可以同 使用与链设置缺省命令行参数 做比较。

`${parameter=default}, ${parameter:=default}`

在没有设置变量的情况下，将其设置为缺省值。

两种形式的作用几乎相同，唯一不同的情况与上面类似，就是当变量 parameter 已经被声明但值为空时。[1]

```
echo ${var=abc}   # abc
echo ${vat=xyz}   # abc
# $var 已经在第一条语句中被赋值为 abc，因此第二条语句将不会改变它的值。
```

${parameter+alt_value}, ${parameter:+alt_value}

如果变量已被设置，使用 alt_value，否则使用空值。

两种形式的作用几乎相同，唯一不同的情况就是当变量 parameter 已经被声明但值为空时，看下面的例子。

```
echo "###### \${parameter+alt_value} ########"
echo

a=${param1+xyz}
echo "a = $a"       # a =

param2=
a=${param2+xyz}
echo "a = $a"       # a = xyz

param3=123
a=${param3+xyz}
echo "a = $a"       # a = xyz

echo
echo "###### \${parameter:+alt_value} ########"
echo

a=${param4:+xyz}
echo "a = $a"       # a =

param5=
a=${param5:+xyz}
echo "a = $a"       # a =
# 不同于 a=${param5+xyz}

param6=123
a=${param6:+xyz}
echo "a = $a"       # a = xyz
```

${parameter?err_msg}, ${parameter:?err_msg}

如果变量已被设置，那么使用原值，否则输出 err_msg 并且终止脚本，返回 错误码 1。

两种形式的作用几乎相同，唯一不同的情况与上面类似，就是当变量 parameter 已经被声明但值为空时。

样例 10-7. 如何使用变量替换和错误信息

```
#!/bin/bash

# 检查系统环境变量。
# 这是一种良好的预防性维护措施。
# 如果控制台用户的名称 $USER 没有被设置，那么主机将不能够识别用户。

: ${HOSTNAME?} ${USER?} ${HOME?} ${MAIL?}
  echo
  echo "Name of the machine is $HOSTNAME."
  echo "You are $USER."
  echo "Your home directory is $HOME."
  echo "Your mail INBOX is located in $MAIL."
  echo
  echo "If you are reading this message,"
  echo "critcial environmental variables have been set."
  echo
  echo

# ------------------------------------------------------

# ${variablename?} 结构统一可以检查脚本中的变量是否被设置。

ThisVariable=Value-of-ThisVariable
# 顺带一提，这个字符串的值可以被设置成名称中不可以使用的禁用字符。
: ${ThisVariable?}
echo "Value of ThisVariable is $ThisVariable."

echo; echo


: ${ZZXy23AB?"ZZXy23AB has not been set."}
# 因为 ZZXy23AB 没有被设置，所以脚本会终止同时显示错误消息。
```

```
# 你可以指定错误消息。
# : ${variablename?"ERROR MESSAGE"}


# 与这些结果相同:  dummy_variable=${ZZXy23AB?}
#                   dummy_variable=${ZZXy23AB?"ZZXy23AB has not be
en set."}
#
#                   echo ${ZZXy23AB?} >/dev/null

# 将上面这些检查变量是否被设置的方法同 "set -u" 作比较。



echo "You will not see this message, because script already term
inated."

HERE=0
exit $HERE   # 将不会从这里退出。

#  事实上，这个脚本将会返回退出码（echo $?）1。
```

样例 10-8. 参数替换与 "usage" 消息

```
#!/bin/bash
# usage-message.sh

: ${1?"Usage: $0 ARGUMENT"}
# 如果命令行参数缺失，脚本将会在这里结束，并且返回下面的错误信息。
#    usage-message.sh: 1: Usage: usage-message.sh ARGUMENT

echo "These two lines echo only if command-line parameter given."

echo "command-line parameter = \"$1\""

exit 0  # 仅当命令行参数存在是才会从这里退出。

# 在传入和未传入命令行参数的情况下查看退出状态。
# 如果传入了命令行参数，那么 "$?" 的结果是0。
# 如果没有，那么 "$?" 的结果是1。
```

参数替换用来处理或扩展变量。下面的表达式是对 `expr` 处理字符串的操作的补足（查看样例 16-9）。这些特殊的表达式通常杀来解析文件的路径名。

变量长度 / 删除子串

`$`

字符串的长度（ `$var` 中字符的个数）。对任意 数组 array， `${#array}` 返回数组中第一个元素的长度。

>  以下情况例外：
>
> - `${#*}` 和 `${#@}` 返回位置参数的个数。
> - 任意数组 array， `${#array[*]}` 和 `${#array[@]}` 返回数组中元素的个数。

样例 10-9. 变量长度

```
#!/bin/bash
# length.sh

E_NO_ARGS=65

if [ $# -eq 0 ]  # 脚本必须传入参数。
then
  echo "Please invoke this script with one or more command-line arguments."
  exit $E_NO_ARGS
fi

var01=abcdEFGH28ij
echo "var01 = ${var01}"
echo "Length of var01 = ${#var01}"
# 现在我们尝试加入空格。
var02="abcd EFGH28ij"
echo "var02 = ${var02}"
echo "Length of var02 = ${#var02}"

echo "Number of command-line arguments passed to script = ${#@}"
echo "Number of command-line arguments passed to script = ${#*}"

exit 0
```

`${var#Pattern}, ${var##Pattern}`

`${var#Pattern}` 删除 `$var` 前缀部分匹配到的最短长度的 `$Pattern` 。

`${var##Pattern}` 删除 `$var` 前缀部分匹配到的最长长度的 `$Pattern` 。

摘自 样例 A-7 的例子：

```
# 函数摘自样例 "day-between.sh"。
# 删除传入的参数中的前缀0。

strip_leading_zero () #  删除传入参数中可能存在的
{                     #+ 前缀0。
  return=${1#0}       #  "1" 代表 "$1"，即传入的参数。
}                     #  从 "$1" 中删除 "0"。
```

下面是由 Manfred Schwarb 提供的上述函数的改进版本：

```
strip_leading_zero2 () # 删除前缀0，
{                      # 否则 Bash 会将其解释为8进制数。
  shopt -s extglob     # 启用扩展通配特性。
  local val=${1##+(0)} # 使用本地变量，匹配前缀中所有的0。
  shopt -u extglob     # 禁用扩展通配特性。
  _strip_leading_zero2=${var:-0}
                       # 如果输入的为0，那么返回 0 而不是 ""。
```

另外一个样例：

```
echo `basename $PWD`        # 当前工作目录的目录名。
echo "${PWD##*/}"           # 当前工作目录的目录名。
echo
echo `basename $0`          # 脚本名。
echo $0                     # 脚本名。
echo "${0##*/}"             # 脚本名。
echo
filename=test.data
echo "${filename##*.}"      # data
                            # 文件扩展名。
```

`${var%Pattern}, ${var%%Pattern}`

`${var%Pattern}` 删除 `$var` 后缀部分匹配到的最短长度的 `$Pattern` 。

`${var%%Pattern}` 删除 `$var` 后缀部分匹配到的最长长度的 `$Pattern` 。

在 Bash 的 第二个版本 中增加了一些额外的选择。

样例 10-10. 参数替换中的模式匹配

```
#!/bin/bash
# patt-matching.sh

# 使用 # ## % %% 参数替换操作符进行模式匹配

var1=abcd12345abc6789
pattern1=a*c  # 通配符 * 可以匹配 a 与 c 之间的任意字符

echo
echo "var1 = $var1"           # abcd12345abc6789
echo "var1 = ${var1}"         # abcd12345abc6789
                              # （另一种形式）
echo "Number of characters in ${var1} = ${#var1}"
echo

echo "pattern1 = $pattern1"   # a*c  (匹配 'a' 与 'c' 之间的一切)
echo "--------------"
echo '${var1#$pattern1}  =' "${var1#$pattern1}"    #         d12
345abc6789
# 匹配到首部最短的3个字符                            abcd1234
5abc6789
#              ^                                     |-|
echo '${var1##$pattern1} =' "${var1##$pattern1}"   #
      6789
# 匹配到首部最长的12个字符                           abcd1234
5abc6789
#              ^                                     |-----
-----|

echo; echo; echo

pattern2=b*9            # 匹配 'b' 与 '9' 之间的任意字符
echo "var1 = $var1"     # 仍旧是 abcd12345abc6789
echo
echo "pattern2 = $pattern2"
echo "--------------"
echo '${var1%pattern2}  =' "${var1%$pattern2}"     #     abcd123
45a
```

```
# 匹配到尾部最短的6个字符                                    abcd12345
abc6789
#             ^
   |----|
echo '${var1%%pattern2} =' "${var1%%$pattern2}"    #     a
# 匹配到尾部最长的12个字符                                   abcd12345
abc6789
#             ^                                                    |-----
--------|

# 牢记 # 与 ## 是从字符串左侧开始，
#      % 与 %% 是从右侧开始。

echo

exit 0
```

样例 10-11. 更改文件扩展名：

```
#!/bin/bash
# rfe.sh: 更改文件扩展名。
#
#         rfe old_extension new_extension
#
# 如：
# 将当前目录下所有 *.gif 文件重命名为 *.jpg，
#         rfe gif jpg


E_BADARGS=65

case $# in
  0|1)             # 竖线 | 在这里表示逻辑或关系。
  echo "Usage: `basename $0` old_file_suffix new_file_suffix"
  exit $E_BADARGS  # 如果只有0个或1个参数，那么退出脚本。
  ;;
esac


for filename in *.$1
# 遍历以第一个参数作为后缀名的文件列表。
do
  mv $filename ${filename%$1}$2
  # 删除文件后缀名，增加第二个参数作为后缀名。
done

exit 0
```

变量扩展 / 替换子串

下面这些结构采用自 ksh。

`${var:pos}`

扩展为从偏移量 pos 处截取的变量 var。

`${var:pos:len}`

扩展为从偏移量 pos 处截取变量 var 最大长度为 len 的字符串。

`${var/Pattern/Replacement}`

替换 var 中第一个匹配到的 Pattern 为 Replacement。

如果 Replacement 被省略，那么匹配到的第一个 Pattern 将被替换为空，即删除。

`${var//Pattern/Replacement}`

全局替换。替换 var 中所有匹配到的 Pattern 为 Replacement。

跟上面一样，如果 Replacement 被省略，那么匹配到的所有 Pattern 将被替换为空，即删除。

样例 10-12. 使用模式匹配解析任意字符串

```
#!/bin/bash

var1=abcd-1234-defg
echo "var1 = $var1"

t=${var1#*-*}
echo "var1 (with everything, up to and including first - stripped out) = $t"
#  t=${var1#*-} 效果相同，
#+ 因为 # 只匹配最短的字符串，
#+ 并且 * 可以任意匹配，其中也包括空字符串。
# （感谢 Stephane Chazelas 指出这一点。）

t=${var##*-*}
echo "If var1 contains a \"-\", returns empty string...   var1 = $t"


t=${var1%*-*}
echo "var1 (with everything from the last - on stripped out) = $t"


echo
```

```
# ------------------------------------------
path_name=/home/bozo/ideas/thoughts/for.today
# ------------------------------------------
echo "path_name = $path_name"
t=${path_name##/*/}
echo "path_name, stripped of prefixes = $t"
# 在这里与 t=`basename $path_name` 效果相同。
#  t=${path_name%/}; t=${t##*/}  是更加通用的方法，
#+ 但有时仍旧也会出现问题。
#  如果 $path_name 以换行结束，那么 `basename $path_name` 将会失效，
#+ 但是上面这种表达式却可以。
# （感谢 S.C.）

t=${path_name%/*.*}
# 同 t=`dirname $path_name` 效果相同。
echo "path_name, stripped of suffixes = $t"
# 在一些情况下会失效，比如 "../", "/foo////", # "foo/", "/"。
#  在删除后缀时，尤其是当文件名没有后缀，目录名却有后缀时，
#+ 事情会变的非常复杂。
# （感谢 S.C.）

echo

t=${path_name:11}
echo "$path_name, with first 11 chars stripped off = $t"
t=${path_name:11:5}
echo "$path_name, with first 11 chars stripped off, length 5 = $t
"

echo

t=${path_name/bozo/clown}
echo "$path_name with \"bozo\" replaced by \"clown\" = $t"
t=${path_name/today/}
echo "$path_name with \"today\" deleted = $t"
t=${path_name//o/O}
echo "$path_name with all o's capitalized = $t"
t=${path_name//o/}
echo "$path_name with all o's deleted = $t"
```

```
exit 0
```

`${var/#Pattern/Replacement}`

替换 var 前缀部分匹配到的 Pattern 为 Replacement。

`${var/%Pattern/Replacement}`

替换 var 后缀部分匹配到的 Pattern 为 Replacement。

样例 10-13. 在字符串首部或尾部进行模式匹配

```
#!/bin/bash
# var-match.sh:
# 演示在字符串首部或尾部进行模式替换。

v0=abc1234zip1234abc    # 初始值。
echo "v0 = $v0"         # abc1234zip1234abc
echo

# 在字符串首部进行匹配
v1=${v0/#abc/ABCDEF}    # abc1234zip123abc
                        # |-|
echo "v1 = $v1"         # ABCDEF1234zip1234abc
                        # |----|

# 在字符串尾部进行匹配
v2=${v0/%abc/ABCDEF}    # abc1234zip123abc
                        #              |-|
echo "v2 = $v2"         # abc1234zip1234ABCDEF
                        #               |----|

echo

#   --------------------------------------------
#   必须在字符串的最开始或者最末尾的地方进行匹配，
#+  否则将不会发生替换。
#   --------------------------------------------
v3=${v0/#123/000}       # 虽然匹配到了，但是不在最开始的地方。
echo "v3 = $v3"         # abc1234zip1234abc
                        # 没有替换。
v4=${v0/%123/000}       # 虽然匹配到了，但是不在最末尾的地方。
echo "v4 = $v4"         # abc1234zip1234abc
                        # 没有替换。

exit 0
```

`${!varprefix*}, ${!varprefix@}`

匹配先前声明过所有以 varprefix 作为变量名前缀的变量。

```
# 这是带 * 或 @ 的间接引用的一种变换形式。
# 在 Bash 2.04 版本中加入了这个特性。

xyz23=whatever
xyz23=

a=${!xyz*}          #  扩展为声明变量中以 "xyz"
# ^ ^   ^              + 开头变量名。
echo "a = $a"       #  a = xyz23 xyz24
a=${!xyz@}          #  同上。
echo "a = $a"       #  a = xyz23 xyz24

echo "---"

abc23=something_else
b=${!abc*}
echo "b = $b"       #  b = abc23
c=${!b}             #  这是我们熟悉的间接引用的形式。
echo $c             #  something_else
```

> 1. 如果在非交互的脚本中，`$parameter` 为空，那么程序将会终止，并且返回 错误码 127（意为“找不到命令”）。↩

第十一章 循环与分支

> 奥赛罗夫人，您为什么把这句话说了又说呢？
>
> —— 《奥赛罗》，莎士比亚

本章目录

- 11.1 循环
- 11.2 嵌套循环
- 11.3 循环控制
- 11.4 测试与分支

对代码块的处理是结构化和构建 shell 脚本的关键。循环与分支结构恰好提供了这样一种对代码块处理的工具。

11.1 循环

循环是当循环控制条件为真时，一系列命令迭代[1]执行的代码块。

for 循环

`for arg in [list]`

这是 shell 中最基本的循环结构，它与C语言形式的循环有着明显的不同。

```
for arg in [list]
do
  command(s)...
done
```

>  在循环的过程中， `arg` 会从 `list` 中连续获得每一个变量的值。
>
> ```
> for arg in "$var1" "$var2" "$var3" ... "$varN"
> # 第一次循环中，arg = $var1
> # 第二次循环中，arg = $var2
> # 第三次循环中，arg = $var3
> # ...
> # 第 N 次循环中，arg = $varN
>
> # 为了防止可能的字符分割问题，[list] 中的参数都需要被引用。
> ```

参数 list 中允许含有 通配符。

如果 `do` 和 `for` 写在同一行时，需要在 list 之后加上一个分号。

`for arg in [list] ; do`

样例 11-1. 简单的 for 循环

```
#!/bin/bash
# 列出太阳系的所有行星。

for planet in Mercury Venus Earth Mars Jupiter Saturn Uranus Nep
tune Pluto
do
  echo $planet  # 每一行输出一个行星。
done

echo; echo

for planet in "Mercury Venus Earth Mars Jupiter Saturn Uranus Ne
ptune Pluto"
    # 所有的行星都输出在一行上。
    # 整个 'list' 被包裹在引号中时是作为一个单一的变量。
    # 为什么？因为空格也是变量的一部分。
do
  echo $planet
done

echo; echo "Whoops! Pluto is no longer a planet!"

exit 0
```

[list] 中的每一个元素中都可能含有多个参数。这在处理参数组中非常有用。在这种情况下，使用 `set` 命令（查看 样例 15-16）强制解析 [list] 中的每一个元素，并将元素的每一个部分分配给位置参数。

样例 11-2. `for` 循环 [list] 中的每一个变量有两个参数的情况

```
#!/bin/bash
# 让行星再躺次枪。

# 将每个行星与其到太阳的距离放在一起。

for planet in "Mercury 36" "Venus 67" "Earth 93" "Mars 142" "Jup
iter 483"
do
  set -- $planet  #  解析变量 "planet"
                  #+ 并将其每个部分赋值给位置参数。
  # "--" 防止一些极端情况，比如 $planet 为空或者以破折号开头。

  # 因为位置参数会被覆盖掉，因此需要先保存原先的位置参数。
  # 你可以使用数组来保存
  #         original_params=("$@")

  echo "$1		$2,000,000 miles from the sum"
  #-------两个制表符---将后面的一系列 0 连到参数 $2 上。
done

# （感谢 S.C. 做出的额外注释。）

exit 0
```

一个单一变量也可以成为 `for` 循环中的 [list]。

样例 11-3. 文件信息：查看一个单一变量中含有的文件列表的文件信息

```
#!/bin/bash
# fileinfo.sh

FILES="/usr/sbin/accept
/usr/sbin/pwck
/usr/sbin/chroot
/usr/bin/fakefile
/sbin/badblocks
/sbin/ypbind"     # 你可能会感兴趣的一系列文件。
                  # 包含一个不存在的文件，/usr/bin/fakefile。

echo

for file in $FILES
do

  if [ ! -e "$file" ]       # 检查文件是否存在。
  then
    echo "$file does not exist."; echo
    continue                # 继续判断下一个文件。
  fi

  ls -l $file | awk '{ print $8 "         file size: " $5 }'  #
输出其中的两个域。
  whatis `basename $file`   # 文件信息。
  # 脚本正常运行需要注意提前设置好 whatis 的数据。
  # 使用 root 权限运行 /usr/bin/makewhatis 可以完成。
  echo
done

exit 0
```

`for` 循环中的 [list] 可以是一个参数。

样例 11-4. 操作含有一系列文件的参数

```
#!/bin/bash

filename="*txt"

for file in $filename
do
 echo "Contents of $file"
 echo "---"
 cat "$file"
 echo
done
```

如果在匹配文件扩展名的 `for` 循环中的 [list] 含有通配符（* 和 ?），那么将会进行文件名扩展。

样例 11-5. 在 `for` 循环中操作文件

```
#!/bin/bash
# list-glob.sh: 通过文件名扩展在 for 循环中产生 [list]。
# 通配 = 文件名扩展。

echo

for file in *
#           ^  Bash 在检测到通配表达式时，
#+             会进行文件名扩展。
do
  ls -l "$file"  # 列出 $PWD（当前工作目录）下的所有文件。
  #  回忆一下，通配符 "*" 会匹配所有的文件名，
  #+ 但是，在文件名扩展中，他将不会匹配以点开头的文件。

  #  如果没有匹配到文件，那么它将会扩展为它自身。
  #  为了防止出现这种情况，需要设置 nullglob 选项。
  #+    (shopt -s nullglob)。
  #  感谢 S.C.
done

echo; echo

for file in [jx]*
do
  rm -f $file    # 删除当前目录下所有以 "j" 或 "x" 开头的文件。
  echo "Removed file \"$file\"".
done

echo

exit 0
```

如果在 `for` 循环中省略 `in [list]` 部分，那么循环将会遍历位置参数（`$@`）。样例 A-15 中使用到了这一点。也可以查看 样例 15-17。

样例 11-6. 缺少 `in [list]` 的 `for` 循环

```
#!/bin/bash

# 尝试在带参数和不带参数两种情况下调用这个脚本，观察发生了什么。

for a
do
 echo -n "$a "
done

#  缺失 'in list' 的情况下，循环会遍历 '$@'
#+（命令行参数列表，包括空格）。

echo

exit 0
```

可以在 `for` 循环中使用 命令代换 生成 [list]。查看 样例 16-54，样例 11-11 和 样例 16-48。

样例 11-7. 在 `for` 循环中使用命令代换生成 [list]

```
#!/bin/bash
# for-loopcmd.sh: 带命令代换所生成 [list] 的 for 循环

NUMBERS="9 7 3 8 37.53"

for number in `echo $NUMBERS`  # for number in 9 7 3 8 37.53
do
  echo -n "$number "
done

echo
exit 0
```

下面是使用命令代换生成 [list] 的更加复杂的例子。

样例 11-8. 一种替代 `grep` 搜索二进制文件的方法

```
#!/bin/bash
# bin-grep.sh: 在二进制文件中定位匹配的字符串。

# 一种替代 `grep` 搜索二进制文件的方法
# 与 "grep -a" 的效果类似

E_BADARGS=65
E_NOFILE=66

if [ $# -ne 2 ]
then
  echo "Usage: `basename $0` search_string filename"
  exit $E_BADARGS
fi

if [ ! -f "$2" ]
then
  echo "File \"$2\" does not exist."
  exit $E_NOFILE
fi


IFS=$'\012'       # 按照 Anton Filippov 的意见应该是
                  # IFS="\n"
for word in $( strings "$2" | grep "$1" )
# "strings" 命令列出二进制文件中的所有字符串。
# 将结果通过管道输出到 "grep" 中，检查是不是匹配的字符串。
do
  echo $word
done

# 就像 S.C. 指出的那样，第 23-30 行可以换成下面的形式：
#    strings "$2" | grep "$1" | tr -s "$IFS" '[\n*]'


# 尝试运行脚本 "./bin-grep.sh mem /bin/ls"

exit 0
```

下面的例子同样展示了如何使用命令代换生成 [list]。

样例 11-9. 列出系统中的所有用户

```
#!/bin/bash
# userlist.sh

PASSWORD_FILE=/etc/passwd
n=1           # 用户数量

for name in $(awk 'BEGIN{fs=":"}{print $1}' < "$PASSWORD_FILE" )
# 分隔符 = :            ^^^^^^
# 输出第一个域                 ^^^^^^^^
# 读取密码文件 /etc/passwd                ^^^^^^^^^^^^^^^^^
do
  echo "USER #$n = $name"
  let "n += 1"
done


# USER #1 = root
# USER #2 = bin
# USER #3 = daemon
# ...
# USER #33 = bozo

exit $?

# 讨论：
# -----
# 一个普通用户是如何读取 /etc/passwd 文件的？
# 提示：检查 /etc/passwd 的文件权限。
# 这算不算是一个安全漏洞？为什么？
```

另外一个关于 [list] 的例子也来自于命令代换。

样例 11-10. 检查目录中所有二进制文件的原作者

```
#!/bin/bash
# findstring.sh
# 在指定目录的二进制文件中寻找指定的字符串。

directory=/usr/bin
fstring="Free Software Foundation"  # 查看哪些文件来自于 FSF。

for file in $( find $directory -type f -name '*' | sort )
do
  strings -f $file | grep "$fstring" | sed -e "s%$driectory%%"
  #  在 "sed" 表达式中，你需要替换掉 "/" 分隔符，
  #+ 因为 "/" 是一个会被过滤的字符。
  #  如果不做替换，将会产生一个错误。（你可以尝试一下。）
done

exit $?

# 简单的练习：
# ----------
# 修改脚本，使其可以从命令行参数中获取 $directory 和 $fstring。
```

最后一个关于 [list] 和命令代换的例子，但这个例子中的命令是一个函数。

```
generate_list ()
{
  echo "one two three"
}

for word in $(generate_list)  # "word" 获得函数执行的结果。
do
  echo "$word"
done

# one
# two
# three
```

`for` 循环的结果可以通过管道导向至一个或多个命令中。

样例 11-11. 列出目录中的所有符号链接。

```
#!/bin/bash
# symlinks.sh: 列出目录中的所有符号链接。

directory=${1-`pwd`}
# 如果没有特别指定，缺省目录为当前工作目录。
# 等价于下面的代码块。
# ----------------------------------------------------
# ARGS=1                  # 只有一个命令行参数。
#
# if [ $# -ne "$ARGS" ]  # 如果不是只有一个参数的情况下
# then
#   directory=`pwd`       # 设为当前工作目录。
# else
#   directory=$1
# fi
# ----------------------------------------------------

echo "symbolic links in directory \"$directory\""

for file in "$( find $directory -type l )"   # -type l = 符号链接
do
  echo "$file"
done | sort                                  # 否则文件顺序会是乱序。

#  严格的来说这里并不需要使用循环，
#+ 因为 "find" 命令的输出结果已经被扩展成一个单一字符串了。
#  然而，为了方便大家理解，我们使用了循环的方式。

#  Dominik 'Aeneas' Schnitzer 指出，
#+ 不引用 $( find $directory -type l ) 的话，
#  脚本将在文件名包含空格时阻塞。

exit 0



# --------------------------------------------------------
# Jean Helou 提供了另外一种方法：
```

```
echo "symbolic links in directory \"$directory\""
# 备份当前的内部字段分隔符。谨慎永远没有坏处。
OLDIFS=$IFS
IFS=:

for file in $(find $directory -type l -printf "%p$IFS")
do      #                              ^^^^^^^^^^^^^^^^
       echo "$file"
       done|sort

# James "Mike" Conley 建议将 Helou 的代码修改为：

OLDIFS=$IFS
IFS='' # 空的内部字段分隔符意味着将不会分隔任何字符串
for file in $( find $directory -type l )
do
  echo $file
  done | sort

#  上面的代码可以在目录名包含冒号（前一个允许包含空格）
#+ 的情况下仍旧正常工作。
```

只需要对上一个样例做一些小小的改动，就可以把在标准输出 `stdout` 中的循环重定向到文件中。

样例 11-12. 将目录中的所有符号链接保存到文件中。

```
#!/bin/bash
# symlinks.sh: 列出目录中的所有符号链接。

OUTFILE=symlinks.list

directory=${1-`pwd`}
# 如果没有特别指定，缺省目录为当前工作目录。


echo "symbolic links in directory \"$directory\"" > "$OUTFILE"
echo "---------------------------" >> "$OUTFILE"

for file in "$( find $directory -type l )"    # -type l = 符号链接

do
  echo "$file"
done | sort >> "$OUTFILE"                     # 将 stdout 的循环结果

#           ^^^^^^^^^^^^^                       重定向到文件。

# echo "Output file = $OUTFILE"

exit $?
```

还有另外一种看起来非常像C语言中循环那样的语法。你需要使用到 双圆括号 语法。

样例 11-13. C语言风格的循环

```
#!/bin/bash
# 用多种方式数到10。

echo

# 基础版
for a in 1 2 3 4 5 6 7 8 9 10
do
  echo -n "$a "
```

```
done

echo; echo

# +==========================================+

# 使用 "seq"
for a in `seq 10`
do
  echo -n "$a "
done

echo; echo

# +==========================================+

# 使用大括号扩展语法
# Bash 3+ 版本有效。
for a in {1..10}
do
  echo -n "$a "
done

echo; echo

# +==========================================+

# 现在用类似C语言的语法再实现一次。

LIMIT=10

for ((a=1; a <= LIMIT ; a++))  # 双圆括号语法，不带 $ 的 LIMIT
do
  echo -n "$a "
done                           # 从 ksh93 中学习到的特性。

echo; echo

# +==========================================+
```

```
# 我们现在使用C语言中的逗号运算符来使得两个变量同时增加。

for ((a=1, b=1; a <= LIMIT ; a++, b++))
do  # 逗号连接操作。
  echo -n "$a-$b "
done

echo; echo

exit 0
```

还可以查看 样例 27-16，样例 27-17 和 样例 A-6。

接下来，我们将展示在真实环境中应用的循环。

样例 11-14. 在批处理模式下使用 `efax`

```
#!/bin/bash
# 传真（必须提前安装了 'efax' 模块）。

EXPECTED_ARGS=2
E_BADARGS=85
MODEM_PORT="/dev/ttyS2"   # 你的电脑可能会不一样。
#                ^^^^^      PCMCIA 调制解调卡缺省端口。

if [ $# -ne $EXPECTED_ARGS ]
# 检查是不是传入了适当数量的命令行参数。
then
   echo "Usage: `basename $0` phone# text-file"
   exit $E_BADARGS
fi


if [ ! -f "$2" ]
then
  echo "File $2 is not a text file."
  #     File 不是一个正常文件或者文件不存在。
  exit $E_BADARGS
```

```
fi


fax make $2              # 根据文本文件创建传真格式文件。

for file in $(ls $2.0*)  # 连接转换后的文件。
                         # 在参数列表中使用通配符（文件名通配）。
do
  fil="$fil $file"
done

efax -d "$MODEM_PORT"  -t "T$1" $fil   # 最后使用 efax。
# 如果上面一行执行失败，尝试添加 -o1。


#  S.C. 指出，上面的 for 循环可以被压缩为
#     efax -d /dev/ttyS2 -o1 -t "T$1" $2.0*
#+ 但是这并不是一个好主意。

exit $?   # efax 同时也会将诊断信息传递给标准输出。
```

>  关键字 `do` 和 `done` 圈定了 for 循环代码块的范围。但是在一些特殊的情况下，也可以被 大括号 取代。
>
> ```
> for((n=1; n<=10; n++))
> # 没有 do！
> {
>   echo -n "* $n *"
> }
> # 没有 done！
>
>
> # 输出：
> # * 1 ** 2 ** 3 ** 4 ** 5 ** 6 ** 7 ** 8 ** 9 ** 10 *
> # 并且 echo $? 返回 0，因此 Bash 并不认为这是一个错误。
>
>
> echo
>
>
> #  但是注意在典型的 for 循环 for n in [list] ... 中，
> #+ 需要在结尾加一个分号。
>
> for n in 1 2 3
> {  echo -n "$n "; }
> #               ^
>
>
> # 感谢 Yongye 指出这一点。
> ```

while 循环

`while` 循环结构会在循环顶部检测循环条件，若循环条件为真（ 退出状态 为0）则循环持续进行。与 `for` 循环 不同的是， `while` 循环是在不知道循环次数的情况下使用的。

```
while [ condition ]
do
  command(s)...
done
```

在 `while` 循环结构中，你不仅可以使用像 `if/test` 中那样的括号结构，也可以使用用途更广泛的双括号结构（`while [[ condition ]]`）。

就像在 `for` 循环中那样，将 `do` 和循环条件放在同一行时需要加一个分号。

`while [ condition ] ; do`

在 `while` 循环中，括号结构并不是必须存在的。比如说 `getopts` 结构。

样例 11-15. 简单的 `while` 循环

```
#!/bin/bash

var0=0
LIMIT=10

while [ "$var0" -lt "$LIMIT" ]
#      ^                    ^
# 必须有空格，因为这是测试结构
do
  echo -n "$var0 "        # -n 不会另起一行
  #             ^           空格用来分开输出的数字。

  var0=`expr $var0 + 1`   # var0=$(($var0+1))  效果相同。
                          # var0=$((var0 + 1)) 效果相同。
                          # let "var0 += 1"    效果相同。
done                      # 还有许多其他的方法也可以达到相同的效果。

echo

exit 0
```

样例 11-16. 另一个例子

```
#!/bin/bash

echo
                                    # 等价于：
while [ "$var1" != "end" ]          # while test "$var1" != "end"
do
  echo "Input variable #1 (end to exit) "
  read var1                         # 不是 'read $var1' （为什么？）。
  echo "variable #1 = $var1"        # 因为存在 "#"，所以需要使用引号。
  # 如果输入的是 "end"，也将会在这里输出。
  # 在结束本轮循环之前都不会再测试循环条件了。
  echo
done

exit 0
```

一个 `while` 循环可以有多个测试条件，但只有最后的那一个条件决定了循环是否终止。这是一种你需要注意到的不同于其他循环的语法。

样例 11-17. 多条件 `while` 循环

```
#!/bin/bash

var1=unset
previous=$var1

while echo "previous-variable = $previous"
      echo
      previous=$var1
      [ "$var1" != end ] # 记录下 $var1 之前的值。
      # 在 while 循环中有4个条件，但只有最后的那个控制循环。
      # 最后一个条件的退出状态才会被记录。
do
echo "Input variable #1 (end to exit) "
  read var1
  echo "variable #1 = $var1"
done

# 猜猜这是怎样实现的。
# 这是一个很小的技巧。

exit 0
```

就像 `for` 循环一样， `while` 循环也可以使用双圆括号结构写得像C语言那样（也可以查看样例 8-5）。

样例 11-18. C语言风格的 `while` 循环

```
#!/bin/bash
# wh-loopc.sh: 在 "while" 循环中计数到10。

LIMIT=10                  # 循环10次。
a=1

while [ "$a" -le $LIMIT ]
do
  echo -n "$a "
  let "a+=1"
done                      # 没什么好奇怪的吧。

echo; echo

# +=================================================+

# 现在我们用C语言风格再写一次。

((a = 1))      # a=1
# 双圆括号结构允许像C语言一样在赋值语句中使用空格。

while (( a <= LIMIT ))    #  双圆括号结构，
do                        #+ 并且没有使用 "$"。
  echo -n "$a "
  ((a += 1))              # let "a+=1"
  # 是的，就是这样。
  # 双圆括号结构允许像C语言一样自增一个变量。
done

echo

# 这可以让C和Java程序猿感觉更加舒服。

exit 0
```

在测试部分， `while` 循环可以调用 函数。

```
t=0

condition ()
{
  ((t++))

  if [ $t -lt 5 ]
  then
    return 0  # true 眞
  else
    return 1  # false 假
  fi
}

while condition
#     ^^^^^^^^^
#     调用函数循环四次。
do
  echo "Still going: t = $t"
done

# Still going: t = 1
# Still going: t = 2
# Still going: t = 3
# Still going: t = 4
```

> 和 if 测试 结构一样，`while` 循环也可以省略括号。
>
> ```
> while condition
> do
>   command(s) ...
> done
> ```

在 `while` 循环中结合 `read` 命令，我们就得到了一个非常易于使用的 `while read` 结构。它可以用来读取和解析文件。

```
cat $filename |     # 从文件获得输入。
while read line     # 只要还有可以读入的行，循环就继续。
do
  ...
done

# =================== 摘自样例脚本 "sd.sh" =================== #

  while read value   # 一次读入一个数据。
  do
    rt=$(echo "scale=$SC; $rt + $value" | bc)
    (( ct++ ))
  done

  am=$(echo "scale=$SC; $rt / $ct" | bc)

  echo $am; return $ct   # 这个功能“返回”了2个值。
  # 注意：这个技巧在 $ct > 255 的情况下会失效。
  # 如果要操作更大的数字，注释掉上面的 "return $ct" 就可以了。
} <"$datafile"   # 传入数据文件。
```

>  在 `while` 循环后面可以通过 < 将标准输入 重定位到文件 中。 `while` 循环同样可以 通过管道 传入标准输入中。

until

与 `while` 循环相反， `until` 循环测试其顶部的循环条件，直到其中的条件为真时停止。

```
until [ condition-is-true ]
do
  commands(s)...
done
```

注意到，跟其他的一些编程语言不同， `until` 循环的测试条件在循环顶部。

就像在 `for` 循环中那样，将 `do` 和循环条件放在同一行时需要加一个分号。

`until[ condition-is-true ] ; do`

样例 11-19. `until` 循环

```
#!/bin/bash

END_CONDITION=end

until [ "$var1" = "$END_CONDITION" ]
# 在循环顶部测试条件。
do
  echo "Input variable #1 "
  echo "($END_CONDITION to exit)"
  read var1
  echo "variable #1 = $var1"
  echo
done

#                 ---                   #

#  就像 "for" 和 "while" 循环一样，
#+ "until" 循环也可以写的像C语言一样。

LIMIT=10
var=0

until (( var > LIMIT ))
do  # ^^ ^     ^     ^^   没有方括号，没有 $ 前缀。
  echo -n "$var "
  (( var++ ))
done    # 0 1 2 3 4 5 6 7 8 9 10


exit 0
```

如何在 `for`，`while` 和 `until` 之间做出选择？我们知道在C语言中，在已知循环次数的情况下更加倾向于使用 `for` 循环。但是在Bash中情况可能更加复杂一些。Bash中的 `for` 循环相比起其他语言来说，结构更加松散，使用更加灵活。因此使用你认为最简单的就好。

1

> 1. 迭代：重复执行一个或一组命令。通常情况下，会使用 `while` 或者 `until` 进行控制。↩

11.2 嵌套循环

嵌套循环，顾名思义就是在循环里面还有循环。外层循环会不断的触发内层循环直到外层循环结束。当然，你仍然可以使用 `break` 可以终止外层或内层的循环。

样例 11-20. 嵌套循环

```
#!/bin/bash
# nested-loop.sh: 嵌套 "for" 循环。

outer=1             # 设置外层循环计数器。

# 外层循环。
for a in 1 2 3 4 5
do
  echo "Pass $outer in outer loop."
  echo "---------------------"
  inner=1           # 重设内层循环计数器。

  # =====================================
  # 内层循环。
  for b in 1 2 3 4 5
  do
    echo "Pass $inner in inner loop."
    let "inner+=1"  # 增加内层循环计数器。
  done
  # 内层循环结束。
  # =====================================

  let "outer+=1"    # 增加外层循环计数器。
  echo              # 在每次外层循环输出中加入空行。
done
# 外层循环结束。

exit 0
```

查看 样例 27-11 详细了解嵌套 while 循环。查看 样例 27-13 详细了解嵌套 until 循环。

11.3 循环控制

> Tournez cent tours, tournez mille tours,
>
> Tournez souvent et tournez toujours . . .
>
> ——保尔·魏尔伦，《木马》

本节介绍两个会影响循环行为的命令。

break, continue

`break` 和 `continue` 命令[1]的作用和在其他编程语言中的作用一样。`break` 用来中止（跳出）循环，而 `continue` 则是略过未执行的循环部分，直接进行下一次循环。

样例 11-21. 循环中 `break` 与 `continue` 的作用

```
#!/bin/bash

LIMIT=19  # 循环上界

echo
echo "Printing Numbers 1 through 20 (but not 3 and 11)."

a=0

while [ $a -le "$LIMIT" ]
do
 a=$(($a+1))

 if [ "$a" -eq 3 ] || [ "$a" -eq 11 ]  # 除了 3 和 11。
 then
   continue      # 略过本次循环的剩余部分。
 fi

 echo -n "$a "   # 当 a 等于 3 和 11 时，将不会执行这条语句。
done
```

```
# 思考：
# 为什么循环不会输出到20？

echo; echo

echo Printing Numbers 1 through 20, but something happens after
2.

##################################################################
##

# 用 'break' 代替了 'continue'。

a=0

while [ "$a" -le "$LIMIT" ]
do
 a=$(($a+1))

 if [ "$a" -gt 2 ]
 then
   break  # 中止循环。
 fi

 echo -n "$a"
done

echo; echo; echo

exit 0
```

`break` 命令接受一个参数。普通的 `break` 命令仅仅跳出其所在的那层循环，而 `break N` 命令则可以跳出其上 N 层的循环。

样例 11-22. 跳出多层循环

```
#!/bin/bash
# break-levels.sh: 跳出循环.

# "break N" 跳出 N 层循环。

for outerloop in 1 2 3 4 5
do
  echo -n "Group $outerloop:   "

  # ------------------------------------------
  for innerloop in 1 2 3 4 5
  do
    echo -n "$innerloop "

    if [ "$innerloop" -eq 3 ]
    then
      break  # 尝试一下 break 2 看看会发生什么。
             # （它同时中止了内层和外层循环。）
    fi
  done
  # ------------------------------------------

  echo
done

echo

exit 0
```

与 `break` 类似，`continue` 也接受一个参数。普通的 `continue` 命令仅仅影响其所在的那层循环，而 `continue N` 命令则可以影响其上 N 层的循环。

样例 11-23. `continue` 影响外层循环

```
#!/bin/bash
# "continue N" 命令可以影响其上 N 层循环。

for outer in I II III IV V           # 外层循环
do
  echo; echo -n "Group $outer: "

  # ------------------------------------------------------------
--------
  for inner in 1 2 3 4 5 6 7 8 9 10  # 内层循环
  do

    if [[ "$inner" -eq 7 && "$outer" = "III" ]]
    then
      continue 2  # 影响两层循环，包括“外层循环”。
                  # 将其替换为普通的 "continue"，那么只会影响内层循环。
    fi

    echo -n "$inner "  # 7 8 9 10 将不会出现在 "Group III."中。
  done
  # ------------------------------------------------------------
--------

done

echo; echo

# 思考：
# 想一个 "continue N" 在脚本中的实际应用情况。

exit 0
```

样例 11-24. 真实环境中的 `continue N`

```
# Albert Reiner 举出了一个如何使用 "continue N" 的例子：
# ---------------------------------------------------

#  如果我有许多任务需要运行，并且运行所需要的数据都以文件的形
#+ 式存在文件夹中。现在有多台设备可以访问这个文件夹，我想将任
```

```
#+ 务分配给这些不同的设备来完成。
#  那么我通常会在每台设备上执行下面的代码：

while true:
do
  for n in .iso.*
  do
    [ "$n" = ".iso.opts" ] && continue
    beta=${n#.iso.}
    [ -r .Iso.$beta ] && continue
    [ -r .lock.$beta ] && sleep 10 && continue
    lockfile -r0 .lock.$beta || continue
    echo -n "$beta: " `date`
    run-isotherm $beta
    date
    ls -alF .Iso.$beta
    [ -r .Iso.$beta ] && rm -rf .lock.$beta
    continue 2
  done
  break
done

exit 0

# 这个脚本中出现的 sleep N 只针对这个脚本，通常的形式是：

while true
do
  for job in {pattern}
  do
    {job already done or running} && continue
    {mark job as running, do job, mark job as done}
    continue 2
  done
  break        # 或者使用类似 `sleep 600` 这样的语句来防止脚本结束。
done

#  这样做可以保证脚本只会在没有任务时（包括在运行过程中添加的任务）
#+ 才会停止。合理使用文件锁保证多台设备可以无重复的并行执行任务（这
#+ 在我的设备上通常会消耗好几个小时，所以我想避免重复计算）。并且，
```

```
#+ 因为每次总是从头开始搜索文件，因此可以通过文件名决定执行的先后
#+ 顺序。当然，你可以不使用 'continue 2' 来完成这些，但是你必须
#+ 添加代码去检测某项任务是否完成（以此判断是否可以执行下一项任务或
#+ 终止、休眠一段时间再执行下一项任务）。
```

>  `continue N` 结构不易理解并且可能在一些情况下有歧义，因此不建议使用。

> 1. 这两个命令是 内建命令，而另外的循环命令，如 `while` 和 `case` 则是 关键词。↩

11.4 测试与分支

`case` 和 `select` 结构并不属于循环结构，因为它们并没有反复执行代码块。但是和循环结构相似的是，它们会根据代码块顶部或尾部的条件控制程序流。

下面介绍两种在代码块中控制程序流的方法：

`case (in)` / `esac`

在 shell 脚本中，`case` 模拟了 C/C++ 语言中的 `switch`，可以根据条件跳转到其中一个分支。其相当于简写版的 `if/then/else` 语句。很适合用来创建菜单选项哟！

```
case "$variable" in
  "$condition1" )
    command...
  ;;
  "$condition2" )
    command...
  ;;
esac
```



- 对变量进行引用不是必须的，因为在这里不会进行字符分割。
- 条件测试语句必须以右括号) 结束。[1]
- 每一段代码块都必须以双分号 ;; 结束。
- 如果测试条件为真，其对应的代码块将被执行，而后整个 `case` 代码段结束执行。
- `case` 代码段必须以 `esac` 结束（倒着拼写case）。

样例 11-25. 如何使用 `case`

```
#!/bin/bash
# 测试字符的种类。

echo; echo "Hit a key, then hit return."
read Keypress

case "$Keypress" in
  [[:lower:]]   ) echo "Lowercase letter";;
  [[:upper:]]   ) echo "Uppercase letter";;
  [0-9]         ) echo "Digit";;
  *             ) echo "Punctuation, whitespace, or other";;
esac      #  字符范围可以用[方括号]表示，也可以用 POSIX 形式的[[双方括号
]]表示。

# 在这个例子的第一个版本中，用来测试是小写还是大写字符使用的是 [a-z] 和 [A
-Z]。
# 这在一些特定的语言环境和 Linux 发行版中不起效。
# POSIX 形式具有更好的兼容性。
# 感谢 Frank Wang 指出这一点。

# 练习：
# -----
# 这个脚本接受一个单字符然后结束。
# 修改脚本，使得其可以循环接受输入，并且检测键入的每一个字符，直到键入 "X"
为止。
# 提示：将所有东西包在 "while" 中。

exit 0
```

样例 11-26. 使用 `case` 创建菜单

```
#!/bin/bash

# 简易的通讯录数据库

clear # 清屏。

echo "          Contact List"
echo "          ------- ----"
```

```
echo "Choose one of the following persons:"
echo
echo "[E]vans, Roland"
echo "[J]ones, Mildred"
echo "[S]mith, Julie"
echo "[Z]ane, Morris"
echo

read person

case "$person" in
# 注意变量是被引用的。

  "E" | "e" )
  # 同时接受大小写的输入。
  echo
  echo "Roland Evans"
  echo "4321 Flash Dr."
  echo "Hardscrabble, CO 80753"
  echo "(303) 734-9874"
  echo "(303) 734-9892 fax"
  echo "revans@zzy.net"
  echo "Business partner & old friend"
  ;;
  # 注意用双分号结束这一个选项。

  "J" | "j" )
  echo
  echo "Mildred Jones"
  echo "249 E. 7th St., Apt. 19"
  echo "New York, NY 10009"
  echo "(212) 533-2814"
  echo "(212) 533-9972 fax"
  echo "milliej@loisaida.com"
  echo "Ex-girlfriend"
  echo "Birthday: Feb. 11"
  ;;

  # Smith 和 Zane 的信息稍后添加。
```

```
  *          )
  # 缺省设置。
  # 空输入（直接键入回车）也是执行这一部分。
  echo
  echo "Not yet in database."
  ;;

esac

echo

# 练习：
# -----
# 修改脚本，使得其可以循环接受多次输入而不是只显示一个地址后终止脚本。

exit 0
```

你可以用 `case` 来检测命令行参数。

```
#!/bin/bash

case "$1" in
  "") echo "Usage: ${0##*/} <filename>"; exit $E_PARAM;;
                      # 没有命令行参数，或者第一个参数为空。
                      # 注意 ${0##*/} 是参数替换 ${var##pattern} 的
一种形式。
                      # 最后的结果是 $0.

  -*) FILENAME=./$1;; #  如果传入的参数以短横线开头，那么将其替换为 ./$1

                      #+ 以避免后续的命令将其解释为一个选项。

  * ) FILENAME=$1;;   # 否则赋值为 $1。
esac
```

下面是一个更加直观的处理命令行参数的例子：

```
#!/bin/bash

while [ $# -gt 0 ]; do    # 遍历完所有参数
  case "$1" in
    -d|--debug)
              # 检测是否是 "-d" 或者 "--debug"。
              DEBUG=1
              ;;
    -c|--conf)
              CONFFILE="$2"
              shift
              if [ ! -f $CONFFILE ]; then
                echo "Error: Supplied file doesn't exist!"
                exit $E_CONFFILE     # 找不到文件。
              fi
              ;;
  esac
  shift       # 检测下一个参数
done

# 摘自 Stefano Falsetto 的 "Log2Rot" 脚本中 "rottlog" 包的一部分。
# 已授权使用。
```

样例 11-27. 使用命令替换生成 `case` 变量

```
#!/bin/bash
# case-cmd.sh: 使用命令替换生成 "case" 变量。

case $( arch ) in   # $( arch ) 返回设备架构。
                    # 等价于 'uname -m"。
  i386 ) echo "80386-based machine";;
  i486 ) echo "80486-based machine";;
  i586 ) echo "Pentium-based machine";;
  i686 ) echo "Pentium2+-based machine";;
  *    ) echo "Other type of machine";;
esac

exit 0
```

`case` 还可以用来做字符串模式匹配。

样例 11-28. 简单的字符串匹配

```
#!/bin/bash
# match-string.sh: 使用 'case' 结构进行简单的字符串匹配。

match_string ()
{ # 字符串精确匹配。
  MATCH=0
  E_NOMATCH=90
  PARAMS=2     # 需要2个参数。
  E_BAD_PARAMS=91

  [ $# -eq $PARAMS ] || return $E_BAD_PARAMS

  case "$1" in
    "$2") return $MATCH;;
    *   ) return $E_NOMATCH;;
  esac

}


a=one
b=two
c=three
d=two

match_string $a     # 参数个数不够
echo $?             # 91

match_string $a $b  # 匹配不到
echo $?             # 90

match_string $a $d  # 匹配成功
echo $?             # 0


exit 0
```

样例 11-29. 检查输入

```
#!/bin/bash
# isaplpha.sh: 使用 "case" 结构检查输入。

SUCCESS=0
FAILURE=1    #  以前是FAILURE=-1,
             #+ 但现在 Bash 不允许返回负值。

isalpha ()  # 测试字符串的第一个字符是否是字母。
{
if [ -z "$1" ]                # 检测是否传入参数。
then
  return $FAILURE
fi

case "$1" in
  [a-zA-Z]*) return $SUCCESS;;  # 是否以字母形式开始?
  *        ) return $FAILURE;;
esac
}             # 可以与 C 语言中的函数 "isalpha ()" 作比较。


isalpha2 ()   # 测试整个字符串是否都是字母。
{
  [ $# -eq 1 ] || return $FAILURE

  case $1 in
  *[!a-zA-Z]*|"") return $FAILURE;;
               *) return $SUCCESS;;
  esac
}

isdigit ()    # 测试整个字符串是否都是数字。
{             # 换句话说,也就是测试是否是一个整型变量。
  [ $# -eq 1 ] || return $FAILURE

  case $1 in
    *[!0-9]*|"") return $FAILURE;;
              *) return $SUCCESS;;
  esac
```

```
}



check_var ()  # 包装后的 isalpha ()。
{
if isalpha "$@"
then
  echo "\"$*\" begins with an alpha character."
  if isalpha2 "$@"
  then        # 其实没必要检查第一个字符是不是字母。
    echo "\"$*\" contains only alpha characters."
  else
    echo "\"$*\" contains at least one non-alpha character."
  fi
else
  echo "\"$*\" begins with a non-alpha character."
              # 如果没有传入参数同样同样返回“存在非字母”。
fi

echo

}

digit_check ()  # 包装后的 isdigit ()。
{
if isdigit "$@"
then
  echo "\"$*\" contains only digits [0 - 9]."
else
  echo "\"$*\" has at least one non-digit character."
fi

echo

}


a=23skidoo
b=H3llo
```

```
c=-What?
d=What?
e=$(echo $b)   # 命令替换。
f=AbcDef
g=27234
h=27a34
i=27.34

check_var $a
check_var $b
check_var $c
check_var $d
check_var $e
check_var $f
check_var      # 如果不传入参数会发送什么？
#
digit_check $g
digit_check $h
digit_check $i


exit 0         # S.C. 改进了本脚本。

# 练习：
# -----
# 写一个函数 'isfloat ()' 来检测输入值是否是浮点数。
# 提示：可以参考函数 'isdigit ()'，在其中加入检测合法的小数点即可。
```

`select`

`select` 结构是学习自 Korn Shell。其同样可以用来构建菜单。

```
select variable [in list]
do
 command...
 break
done
```

而效果则是终端会提示用户输入列表中的一个选项。注意，`select` 默认使用提示字串3（Prompt String 3，`$PS3`，即#?），但同样可以被修改。

样例 11-30. 使用 `select` 创建菜单

```
#!/bin/bash

PS3='Choose your favorite vegetable: ' # 设置提示字串。
                                       # 否则默认为 #?。

echo

select vegetable in "beans" "carrots" "potatoes" "onions" "rutab
agas"
do
  echo
  echo "Your favorite veggie is $vegetable."
  echo "Yuck!"
  echo
  break  # 如果没有 'break' 会发生什么？
done

exit

# 练习：
# -----
# 修改脚本，使得其可以接受其他输入而不是 "select" 语句中所指定的。
# 例如，如果用户输入 "peas,"，那么脚本会通知用户 "Sorry. That is not o
n the menu."
```

如果 *in list* 被省略，那么 `select` 将会使用传入脚本的命令行参数（`$@`）或者传入函数的参数作为 *list*。

可以与 `for variable [in list]` 中 *in list* 被省略的情况做比较。

样例 11-31. 在函数中使用 `select` 创建菜单

```
#!/bin/bash

PS3='Choose your favorite vegetable: '

echo

choice_of()
{
select vegetable
# [in list] 被省略，因此 'select' 将会使用传入函数的参数作为 list。
do
  echo
  echo "Your favorite veggie is $vegetable."
  echo "Yuck!"
  echo
  break
done
}

choice_of beans rice carrorts radishes rutabaga spinach
#         $1    $2   $3       $4       $5       $6
#         传入了函数 choice_of()

exit 0
```

还可以参照 样例37-3。

1

1. 在写匹配行的时候，可以在左边加上左括号 (，使整个结构看起来更加优雅。

```
case $( arch ) in   # $( arch ) 返回设备架构。
  ( i386 ) echo "80386-based machine";;
# ^      ^
  ( i486 ) echo "80486-based machine";;
  ( i586 ) echo "Pentium-based machine";;
  ( i686 ) echo "Pentium2+-based machine";;
  (    * ) echo "Other type of machine";;
esac
```



第十二章 命令替换

命令替换重新指定一个[1]或多个命令的输出。其实就是将命令的输出导到另外一个地方[2]。

命令替换的通常形式是（ `` `...` `` ），即用反引号引用命令。

```
script_name=`basename $0`
echo "The name of this script is $scirpt_name."
```

命令的输出可以作为另一个命令的参数，也可以赋值给一个变量。甚至在 `for` 循环中可以用输出产生参数表。

```
rm `cat filename`   # "filename" 中包含了一系列需要被删除的文件名。
#
# S.C. 指出这样写可能会导致出现 "arg list too long" 的错误。
# 更好的写法应该是 xargs rm -- < filename
# （ -- 可以在 "filename" 文件名以 "-" 为开头时仍旧正常执行 ）

textfile_listing=`ls *.txt`
# 变量中包含了当前工作目录下所有的名为 *.txt 的文件。
echo $textfile_listing

textfile_listing2=$(ls *.txt)   # 命令替换的另一种形式。
echo $textfile_listing2
# 结果相同。

# 这样将一系列文件名赋值给一个单一字符串可能会出现换行。
#
# 而更加安全的方式是将这一系列文件存入数组。
#      shopt -s nullglob    # 设置后，如果没有匹配到文件，那么变量会被赋值为空。
#      textfile_listing=( *.txt )
#
# 感谢 S.C.
```

 命令替换本质上是调用了一个 子进程 来执行。

 命令替换有可能会出现 字符分割 的情况。

```
COMMAND `echo a b`     # 2个参数：a和b

COMMAND "`echo a b`"   # 1个参数："a b"

COMMAND `echo`         # 没有参数

COMMAND "`echo`"       # 一个空参数


# 感谢 S.C.
```

但即使不存在字符分割的情况，使用命令替换也会出现丢失尾部换行符的情况。

```
# cd "`pwd`"  # 你是不是认为这条语句在任何情况下都不会出现错误？
# 但事实却不是这样的。

mkdir 'dir with trailing newline
'

cd 'dir with trailing newline
'

cd "`pwd`"  # Bash 会出现如下错误提示：
# bash: cd: /tmp/file with trailing newline: No such file or
 directory

cd "$PWD"   # 这样写是对的。
```

```
old_tty_setting=$(stty -g)   # 保存旧的设置。
echo "Hit a key "
stty -icanon -echo           # 禁用终端的 canonical 模式。
                             # 同时禁用 echo。
key=$(dd bs=1 count=1 2> /dev/null)   # 使用 'dd' 获得键值。
stty "$old_tty_setting"      # 恢复旧的设置。
echo "You hit ${#key} key."  # ${#variable} 表示 $variable 中
的字符个数。
#
# 除了按下回车键外，其余情况都会输出 "You hit 1 key."
# 按下回车键会输出 "You hit 0 key."
# 因为唯一的换行符在命令替换中被丢失了。

# 这段代码摘自 Stéphane Chazelas。
```

 使用 `echo` 输出未被引用的命令代换的变量时会删掉尾部的换行。这可能会导致非常不好的情况出现。

```
dir_listing=`ls -l`
echo $dir_listing     # 未被引用

# 你希望会出现按行显示出文件列表。

# 但是，你却看到了：
# total 3 -rw-rw-r-- 1 bozo bozo 30 May 13 17:15 1.txt -rw-r
w-r-- 1 bozo
# bozo 51 May 15 20:57 t2.sh -rwxr-xr-x 1 bozo bozo 217 Mar
5 21:13 wi.sh

# 所有换行都消失了。


echo "$dir_listing"   # 被引用
# -rw-rw-r--    1 bozo       30 May 13 17:15 1.txt
# -rw-rw-r--    1 bozo       51 May 15 20:57 t2.sh
# -rwxr-xr-x    1 bozo      217 Mar  5 21:13 wi.sh
```

你甚至可以使用 重定向 或者 `cat` 命令把一个文件的内容通过命令代换赋值给一个变量。

```
variable1=`<file1`      # 将 "file1" 的内容赋值给 variable1。
variable2=`cat file2`   # 将 "file2" 的内容赋值给 variable2。
                        # 使用 cat 命令会开一个新进程，因此执行速度会比
重定向慢。

# 需要注意的是，这些变量中可能包含一些空格或者控制字符。

# 无需显示的赋值给一个变量。
echo "` <$0`"           # 输出脚本自身。
```

```
#  摘录自系统文件 /etc/rc.d/rc.sysinit
#+ （Red Hat Linux 发行版）


if [ -f /fsckoptions ]; then
        fsckoptions=`cat /fsckoptions`
...
fi
#
#
if [ -e "/proc/ide/${disk[$device]}/media" ] ; then
             hdmedia=`cat /proc/ide/${disk[$device]}/media`
...
fi
#
#
if [ ! -n "`uname -r | grep -- "-"`" ]; then
       ktag="`cat /proc/version`"
...
fi
#
#
if [ $usb = "1" ]; then
    sleep 5
    mouseoutput=`cat /proc/bus/usb/devices 2>/dev/null|grep -E "
^I.*Cls=03.*Prot=02"`
    kbdoutput=`cat /proc/bus/usb/devices 2>/dev/null|grep -E "^I
.*Cls=03.*Prot=01"`
...
fi
```

> ⚠ 尽量不要将一大段文字赋值给一个变量，除非你有足够的理由。也绝不要将一个二进制文件的内容赋值给一个变量。
>
> 样例 12-1. 蠢蠢的脚本
>
> ```
> #!/bin/bash
> # stupid-script-tricks.sh: 不要在自己的电脑上尝试。
> # 摘自 "Stupid Script Tricks" 卷一。
>
> exit 99  ### 如果你有胆，就注释掉这行。:)
>
> dangerous_variable=`cat /boot/vmlinuz`   # 压缩的 Linux 内核。
>
> echo "string-length of \$dangerous_variable = ${#dangerous_variable}"
> # $dangerous_variable 的长度为 794151
> # （更新版本的内核可能更大。）
> # 与 'wc -c /boot/vmlinuz' 的结果不同。
>
> # echo "$dangerous_variable"
> # 不要作死。否则脚本会挂起。
>
>
>
> # 将二进制文件的内容赋值给一个变量没有任何意义。
>
> exit 0
> ```
>
> 尽管脚本会挂起，但并不会出现缓存溢出的情况。而这正是像 Bash 这样的解释型语言相比起编译型语言能够提供更多保护的一个例子。

命令替换允许将 循环 的输出结果赋值给一个变量。这其中的关键在于循环内部的 `echo` 命令。

样例 12-2. 将循环的输出结果赋值给变量

```
#!/bin/bash
# csubloop.sh: 将循环的输出结果赋值给变量。

variable1=`for i in 1 2 3 4 5
do
  echo -n "$i"                 #  在这里，'echo' 命令非常关键。
done`

echo "variable1 = $variable1"  # variable1 = 12345


i=0
variable2=`while [ "$i" -lt 10 ]
do
  echo -n "$i"                 # 很关键的 'echo'。
  let "i += 1"                 # i 自增。
done`

echo "variable2 = $variable2"  # variable2 = 0123456789

# 这个例子表明可以在变量声明时嵌入循环。

exit 0
```

> 命令替换能够让 Bash 做更多的事情。而这仅仅需要在书写程序或者脚本时将结果输出到标准输出 `stdout` 中，然后将这些输出结果赋值给变量即可。
>
> ```
> #include <stdio.h>
>
> /*  "Hello, world." C program  */
>
> int main()
> {
>   printf( "Hello, world.\n" );
>   return (0);
> }
> ```

```
bash$ gcc -0 hello hello.c
```

```
#!/bin/bash
# hello.sh

greeting=`./hello`
echo $greeting
```

```
bash$ sh hello.sh
Hello, world.
```

 在命令替换中，你可以使用 `$(...)` 来替代反引号。

```
output=$(sed -n /"$1"/p $file)   # 摘自 "grp.sh"。

# 将文本文件的内容赋值给一个变量。
File_contents1=$(cat $file1)
File_contents2=$(<$file2)        # 这么做也是可以的。
```

`$(...)` 和反引号在处理双反斜杠上有所不同。

```
bash$ echo `echo \\`


bash$ echo $(echo \\)
\
```

`$(...)` 允许嵌套。[3]

```
word_count=$( wc -w $(echo * | awk '{print $8}') )
```

样例 12-3. 寻找变位词（anagram）

```
#!/bin/bash
```

```
# agram2.sh
# 嵌套命令替换的例子。

# 其中使用了作者写的工具包 "yawl" 中的 "anagram" 工具。
# http://ibiblio.org/pub/Linux/libs/yawl-0.3.2.tar.gz
# http://bash.deta.in/yawl-0.3.2.tar.gz

E_NOARGS=86
E_BADARG=87
MINLEN=7

if [ -z "$1" ]
then
  echo "Usage $0 LETTERSET"
  exit $E_NOARGS         # 脚本需要命令行参数。
elif [ ${#1} -lt $MINLEN ]
then
  echo "Argument must have at least $MINLEN letters."
  exit $E_BADARG
fi



FILTER='.......'         # 至少需要7个字符。
#       1234567
Anagrams=( $(echo $(anagram $1 | grep $FILTER) ) )
#          $(     $(        嵌套命令集        ) )
#        (                 赋值给数组                 )

echo
echo "${#Anagrams[*]}  7+ letter anagrams found"
echo
echo ${Anagrams[0]}      # 第一个变位词。
echo ${Anagrams[1]}      # 第二个变位词。
                         # 以此类推。

# echo "${Anagrams[*]}"  # 将所有变位词在一行里面输出。

# 可以配合后面的数组章节来理解上面的代码。
```

```
# 建议同时查看另一个寻找变位词的脚本 agram.sh。

exit $?
```

以下是包含命令替换的样例：

1. 样例 11-8
2. 样例 11-27
3. 样例 9-16
4. 样例 16-3
5. 样例 16-22
6. 样例 16-17
7. 样例 16-54
8. 样例 11-14
9. 样例 11-11
10. 样例 16-32
11. 样例 20-8
12. 样例 A-16
13. 样例 29-3
14. 样例 16-47
15. 样例 16-48
16. 样例 16-49

> [1]. 在命令替换中可以使用外部系统命令，内建命令 甚至是 脚本函数。↩

> [2]. 从技术的角度来讲，命令替换实际上是获得了命令输出到标准输出的结果，然后通过赋值号将结果赋值给一个变量。↩

> [3]. 事实上，使用反引号进行嵌套也是可行的。但是 John Default 提醒到需要将内部的反引号进行转义。
>
> ```
> word_count=\` wc -w \\\`echo * | awk '{print $8}'\\\`
> \`
> ```
>
> ↩

第十三章 算术扩展

算术扩展为脚本中的（整数）算术操作提供了强有力的工具。你可以使用反引号、双圆括号或者 `let` 将字符串转换为数学表达式。

差异比较

使用 反引号 的算术扩展（通常与 `expr` 一起使用）

```
z=`expr $z + 3`          # 'expr' 命令执行了算术扩展。
```

使用 双圆括号 或 `let` 的算术扩展。

事实上，在算术扩展中，反引号已经被双圆括号 `((...))` 和 `$((...))` 以及 `let` 所取代。

```
z=$(($z+3))
z=$((z+3))                          # 同样正确。
                                    # 在双圆括号内，参数引用形式可用可不用。



# $((EXPRESSION)) 是算术扩展。   # 不要与命令替换混淆。



# 双圆括号不是只能用作赋值算术结果。

  n=0
  echo "n = $n"                     # n = 0

  (( n += 1 ))                      # 自增。
# (( $n += 1 )) 是错误用法！
  echo "n = $n"                     # n = 1


let z=z+3
let "z += 3"  # 引号允许在赋值表达式中使用空格。
              # 'let' 事实上执行的算术运算而非算术扩展。
```

以下是包含算术扩展的样例：

1. 样例 16-9
2. 样例 11-15
3. 样例 27-1
4. 样例 27-11
5. 样例 A-16

第十四章 休息时间

作者开始玩不转不是外国人的游戏了。亲爱的读者可以藉此休息一下，如果可以，请帮助我们推广一下本书原作和译作。

原作作者致所有读者

各位 Linux 用户，你们好！你们现在正阅读的这本书能够给你们带来好运。

所以赶紧打开你们的邮箱，将本文的访问链接发给你的10位朋友。

但是在发邮件之前，记得粘贴一段大约100行的 Bash 脚本在邮件后面。

千万不要打断这个传递，并且一定要在48小时内发送邮件！

布鲁克林区的 Wilfred P. 没有发出10封邮件。当他第三天起床时发现他变成了一名 COBOL 程序员。

纽波特纽斯港的 Howard L. 按时发出了10封邮件。然后一个月内，他就有了足够的硬件来搭建一个100个节点的 Beowulf 集群来玩 Tuxracer。

芝加哥的 Amelia V. 看到以后不屑一顾，置之不理。不久之后，她的终端炸了。现在她不得不为微软工作，撰写文档。

千万不要打断这个传递！马上去发邮件吧！

> Courtesy 'NIX "fortune cookies", with some alterations and many apologies

第五部分 高级话题

目录

- 18.正则表达式
  - 18.1正则表达式简介
  - 18.2文件名替换
- 19. 嵌入文档
- 20. I/O 重定向
  - 20.1 使用 exec
  - 20.2 重定向代码块
  - 20.3 应用程序
- 22. 限制模式的Shell
- 24. 函数
  - 24.1 复杂函数和函数复杂性
  - 24.2 局部变量
  - 24.3 不适用局部变量的递归
- 25. 别名
- 27. 数组
- 30. 网络编程
- 33. 选项
- 34. 陷阱
- 38. 后记
  - 38.1 作者后记
  - 38.2 关于作者
  - 38.3 从哪里可以获得帮助
  - 38.4 用来制作这本书的工具
  - 38.5 致谢
  - 38.6 免责声明

19 嵌入文档

> Here and now, boys. --Aldous Huxley, Island

嵌入文档是一段有特殊作用的代码块，它用 I/O 重定向 在交互程序和交互命令中传递和反馈一个命令列表，例如 ftp，cat 或者是 ex 文本编辑器

```
COMMAND <<InputComesFromHERE
...
...
...
InputComesFromHERE
```

嵌入文档用限定符作为命令列表的边界，在限定符前需要一个指定的标识符 `<<` ，这会将一个程序或命令的标准输入(stdin)进行重定向，它类似 `交互程序 < 命令文件` 的方式，其中命令文件内容如下

```
command #1
command #2
...
```

嵌入文档的格式大致如下

```
interactive-program <<LimitString
command #1
command #2
...
LimitString
```

限定符的选择必须保证特殊以确保不会和命令列表里的内容发生混淆。

注意嵌入文档有时候用作非交互的工具和命令有着非常好的效果，例如 wall

样例 19-1. broadcast: 给每个登陆者发送信息

```
#!/bin/bash

wall <<zzz23EndOfMessagezzz23
E-mail your noontime orders for pizza to the system administrato
r.
    (Add an extra dollar for anchovy or mushroom topping.)
# 额外的信息文本.
# 注意: 'wall' 会打印注释行.
zzz23EndOfMessagezzz23

# 更有效的做法是通过
#         wall < 信息文本
#  然而, 在脚本里嵌入信息模板不乏是一种迅速而又随性的解决方式.

exit
```

样例: 19-2. dummyfile：创建一个有两行内容的虚拟文件

```
#!/bin/bash

# 非交互的使用 `vi` 编辑文件.
# 仿照 'sed'.

E_BADARGS=85

if [ -z "$1" ]
then
  echo "Usage: `basename $0` filename"
  exit $E_BADARGS
fi

TARGETFILE=$1

# 插入两行到文件中保存
#--------Begin here document-----------#
vi $TARGETFILE <<x23LimitStringx23
i
This is line 1 of the example file.
This is line 2 of the example file.
^[
ZZ
x23LimitStringx23
#----------End here document-----------#

#  注意 "^" 对 "[" 进行了转义
#+ 这段起到了和键盘上按下 Control-V <Esc> 相同的效果.

#  Bram Moolenaar 指出这种情况下 'vim' 可能无法正常工作
#+ 因为在与终端交互的过程中可能会出现问题.

exit
```

上述脚本实现了 `ex` 的功能, 而不是 `vi` . 嵌入文档包含了 `ex` 足够通用的命令列表来形成自有的类别, 所以又称之为 `ex` 脚本.

```
#!/bin/bash
#  替换所有的以 ".txt" 后缀结尾的文件的 "Smith" 为 "Jones"

ORIGINAL=Smith
REPLACEMENT=Jones

for word in $(fgrep -l $ORIGINAL *.txt)
do
  # -------------------------------------
  ex $word <<EOF
  :%s/$ORIGINAL/$REPLACEMENT/g
  :wq
EOF
  # :%s is the "ex" substitution command.
  # :wq is write-and-quit.
  # -------------------------------------
done
```

类似的 `ex 脚本` 是 `cat 脚本`.

样例 19-3. 使用 `cat` 的多行信息

```
#!/bin/bash

#  'echo' 可以输出单行信息,
#+  但是如果是输出消息块就有点问题了.
#   'cat' 嵌入文档却能解决这个局限.

cat <<End-of-message
-------------------------------------
This is line 1 of the message.
This is line 2 of the message.
This is line 3 of the message.
This is line 4 of the message.
This is the last line of the message.
-------------------------------------
End-of-message

#  替换上述嵌入文档内的 7 行文本
#+   cat > $Newfile <<End-of-message
#+       ^^^^^^^^^^
#+ 将输出追加到 $Newfile, 而不是标准输出.

exit 0



#--------------------------------------------
# 由于上面的 "exit 0"，下面的代码将不会生效.

# S.C. points out that the following also works.
echo "-------------------------------------
This is line 1 of the message.
This is line 2 of the message.
This is line 3 of the message.
This is line 4 of the message.
This is the last line of the message.
-------------------------------------"
# 然而, 文本可能不包括双引号除非出现了字符串逃逸.
```

`-` 的作用是标记了一个嵌入文档限制符 (<<-LimitString) ，它能抑制输出的行首的 `tab` (非空格). 这在脚本可读性方面可能非常有用.

样例 19-4. 抑制 tab 的多行信息

```
#!/bin/bash
# 和之前的样例一样, 但...

#  嵌入文档内的 '-' ，也就是 <<-
#+ 抑制了文档行首的 'tab',
#+ 但 *不是* 空格.

cat <<-ENDOFMESSAGE
    This is line 1 of the message.
    This is line 2 of the message.
    This is line 3 of the message.
    This is line 4 of the message.
    This is the last line of the message.
ENDOFMESSAGE
# 脚本的输出将左对齐.
# 行首的 tab 将不会输出.

# 上面 5 行的 "信息" 以 tab 开始, 不是空格.
# 空格不会受影响 <<- .

# 注意这个选项对 *内嵌的* tab 没有影响.

exit 0
```

嵌入文档支持参数和命令替换. 因此可以向嵌入文档传递不同的参数,变向的改其输出.

样例 19-5. 可替换参数的嵌入文档

```
#!/bin/bash
# 另一个使用参数替换的 'cat' 嵌入文档.

# 试一试没有命令行参数,   ./scriptname
# 试一试一个命令行参数,   ./scriptname Mortimer
# 试试用一两个单词引用命令行参数,
#                           ./scriptname "Mortimer Jones"

CMDLINEPARAM=1     #  Expect at least command-line parameter.

if [ $# -ge $CMDLINEPARAM ]
then
  NAME=$1          #  If more than one command-line param,
                   #+ then just take the first.
else
  NAME="John Doe"  #  Default, if no command-line parameter.
fi

RESPONDENT="the author of this fine script"


cat <<Endofmessage

Hello, there, $NAME.
Greetings to you, $NAME, from $RESPONDENT.

# 这个注释在输出时显示 (为什么?).

Endofmessage

# 注意输出了空行.
# 所以可以这样注释.

exit
```

这个包含参数替换的嵌入文档是相当有用的

样例 19-6. 上传文件对到 `Sunsite` 入口目录

```
#!/bin/bash
# upload.sh

#  上传文件对 (Filename.lsm, Filename.tar.gz)
#+ 到 Sunsite/UNC (ibiblio.org) 的入口目录.
#  Filename.tar.gz 是个 tarball.
#  Filename.lsm is 是个描述文件.
#  Sunsite 需要 "lsm" 文件, 否则将会退回给发送者


E_ARGERROR=85

if [ -z "$1" ]
then
  echo "Usage: `basename $0` Filename-to-upload"
  exit $E_ARGERROR
fi


Filename=`basename $1`           # Strips pathname out of file n
ame.

Server="ibiblio.org"
Directory="/incoming/Linux"
#  脚本里不需要硬编码,
#+ 但最好可以替换命令行参数.

Password="your.e-mail.address"   # Change above to suit.

ftp -n $Server <<End-Of-Session
# -n 禁用自动登录

user anonymous "$Password"       #  If this doesn't work, then t
ry:
                                 #  quote user anonymous "$Passw
ord"
binary
bell                             # Ring 'bell' after each file t
ransfer.
```

```
cd $Directory
put "$Filename.lsm"
put "$Filename.tar.gz"
bye
End-Of-Session

exit 0
```

在嵌入文档头部引用或转义"限制符"来禁用参数替换.原因是 `引用/转义` 限定符能有效的转义 "$", "`", 和 "\" 这些特殊符号, 使他们维持字面上的意思. (感谢 Allen Halsey 指出这点.)

样例 19-7. 禁用参数替换

```
#!/bin/bash
#  A 'cat' here-document, but with parameter substitution disabl
ed.

NAME="John Doe"
RESPONDENT="the author of this fine script"

cat <<'Endofmessage'

Hello, there, $NAME.
Greetings to you, $NAME, from $RESPONDENT.

Endofmessage

#   当'限制符'引用或转义时不会有参数替换.
#   下面的嵌入文档也有同样的效果
#   cat <<"Endofmessage"
#   cat <<\Endofmessage



#   同样的:

cat <<"SpecialCharTest"
```

```
Directory listing would follow
if limit string were not quoted.
`ls -l`

Arithmetic expansion would take place
if limit string were not quoted.
$((5 + 3))

A a single backslash would echo
if limit string were not quoted.
\\

SpecialCharTest


exit
```

生成脚本或者程序代码时可以用禁用参数的方式来输出文本.

样例 19-8. 生成其他脚本的脚本

```
#!/bin/bash
# generate-script.sh
# Based on an idea by Albert Reiner.

OUTFILE=generated.sh         # Name of the file to generate.



# -----------------------------------------------------------
# '嵌入文档涵盖了生成脚本的主体部分.
(
cat <<'EOF'
#!/bin/bash

echo "This is a generated shell script."
#  注意我们现在在一个子 shell 内,
#+ 我们不能访问 "外部" 脚本变量.

echo "Generated file will be named: $OUTFILE"
#  上面这行并不能按照预期的正常工作
```

```
#+ 因为参数扩展已被禁用.
#  相反的, 结果是文字输出.

a=7
b=3

let "c = $a * $b"
echo "c = $c"

exit 0
EOF
) > $OUTFILE
# -----------------------------------------------------------

#  在上述的嵌入文档内引用'限制符'防止变量扩展

if [ -f "$OUTFILE" ]
then
  chmod 755 $OUTFILE
  # 生成可执行文件.
else
  echo "Problem in creating file: \"$OUTFILE\""
fi

#  这个方法适用于生成 C, Perl, Python, Makefiles 等等

exit 0
```

可以从嵌入文档的输出设置一个变量的值. 这实际上是种灵活的 命令替换.

```
variable=$(cat <<SETVAR
This variable
runs over multiple lines.
SETVAR
)

echo "$variable"
```

同样的脚本里嵌入文档可以作为函数的输入.

样例 19-9. 嵌入文档和函数

```
#!/bin/bash
# here-function.sh

GetPersonalData ()
{
  read firstname
  read lastname
  read address
  read city
  read state
  read zipcode
} # 可以肯定的是这应该是个交互式的函数, 但 . . .


# 作为函数的输入.
GetPersonalData <<RECORD001
Bozo
Bozeman
2726 Nondescript Dr.
Bozeman
MT
21226
RECORD001


echo
echo "$firstname $lastname"
echo "$address"
echo "$city, $state $zipcode"
echo

exit 0
```

可以这样使用: 作为一个虚构的命令接受嵌入文档的输出. 这样实际上就创建了一个"匿名" 嵌入文档.

样例 19-10. "匿名" 嵌入文档

```
#!/bin/bash

: <<TESTVARIABLES
${HOSTNAME?}${USER?}${MAIL?}  # Print error message if one of th
e variables not set.
TESTVARIABLES

exit $?
```

- 上面技巧的一种变体允许"可添加注释"的代码块.

样例 19-11. 可添加注释的代码块

```
#!/bin/bash
# commentblock.sh

: <<COMMENTBLOCK
echo "This line will not echo."
这些注释没有 "#" 前缀.
则是另一种没有 "#" 前缀的注释方法.

&*@!!++=
上面这行不会产生报错信息,
因为 bash 解释器会忽略它.

COMMENTBLOCK

echo "Exit value of above \"COMMENTBLOCK\" is $?."   # 0
# 没有错误输出.
echo

#  上面的技巧经常用于工作代码的注释用作排错目的
#  这省去了在每一行开头加上 "#" 前缀,
#+ 然后调试完不得不删除每行的前缀的重复工作.
#  注意我们用了 ":", 在这之上，是可选的.

echo "Just before commented-out code block."
#  下面这个在双破折号之间的代码不会被执行.
#  ==============================================================
```

```
======
: <<DEBUGXXX
for file in *
do
 cat "$file"
done
DEBUGXXX
#  =============================================================
======
echo "Just after commented-out code block."

exit 0




#################################################################
######
#  注意, 然而, 如果将变量中包含一个注释的代码块将会引发问题
#  例如:


#/!/bin/bash

  : <<COMMENTBLOCK
  echo "This line will not echo."
  &*@!!++=
  ${foo_bar_bazz?}
  $(rm -rf /tmp/foobar/)
  $(touch my_build_directory/cups/Makefile)
COMMENTBLOCK


$ sh commented-bad.sh
commented-bad.sh: line 3: foo_bar_bazz: parameter null or not se
t

# 有效的补救办法就是在 49 行的位置加上单引号，变为 'COMMENTBLOCK'.

  : <<'COMMENTBLOCK'
```

```
# 感谢 Kurt Pfeifle 指出这一点.
```

- 另一个漂亮的方法使得"自文档化"的脚本成为可能

样例 19-12. 自文档化的脚本

```
#!/bin/bash
# self-document.sh: self-documenting script
# Modification of "colm.sh".

DOC_REQUEST=70

if [ "$1" = "-h"  -o "$1" = "--help" ]     # 请求帮助.
then
  echo; echo "Usage: $0 [directory-name]"; echo
  sed --silent -e '/DOCUMENTATIONXX$/,/^DOCUMENTATIONXX$/p' "$0"
 |
  sed -e '/DOCUMENTATIONXX$/d'; exit $DOC_REQUEST; fi


: <<DOCUMENTATIONXX
List the statistics of a specified directory in tabular format.
---------------------------------------------------------------
The command-line parameter gives the directory to be listed.
If no directory specified or directory specified cannot be read,
then list the current working directory.

DOCUMENTATIONXX

if [ -z "$1" -o ! -r "$1" ]
then
  directory=.
else
  directory="$1"
fi

echo "Listing of "$directory":"; echo
(printf "PERMISSIONS LINKS OWNER GROUP SIZE MONTH DAY HH:MM PROG
-NAME\n" \
; ls -l "$directory" | sed 1d) | column -t

exit 0
```

使用 cat script 是另一种可行的方法.

```
DOC_REQUEST=70

if [ "$1" = "-h"  -o "$1" = "--help" ]     # Request help.
then                                        # Use a "cat script"
. . .
  cat <<DOCUMENTATIONXX
List the statistics of a specified directory in tabular format.
---------------------------------------------------------------
The command-line parameter gives the directory to be listed.
If no directory specified or directory specified cannot be read,
then list the current working directory.

DOCUMENTATIONXX
exit $DOC_REQUEST
fi
```

> 另请参阅 样例 A-28, 样例 A-40, 样例 A-41, and 样例 A-42 更多样例请阅读脚本附带的注释文档.

- 嵌入文档创建了临时文件, 但这些文件在打开且不可被其他程序访问后删除.

```
bash$ bash -c 'lsof -a -p $$ -d0' << EOF
> EOF
lsof    1213 bozo    0r    REG    3,5    0 30386 /tmp/t1213-0-sh
(deleted)
```

- 某些工具在嵌入文档内部并不能正常运行.
- 在嵌入文档的最后关闭限定符必须在起始的第一个字符的位置开始.行首不能是空格. 限制符后尾随空格同样会导致意想不到的行为.空格可以防止限制符被当做其他用途. [1]

```
#!/bin/bash

echo "-----------------------------------------------------------
------------"

cat <<LimitString
echo "This is line 1 of the message inside the here document."
echo "This is line 2 of the message inside the here document."
echo "This is the final line of the message inside the here docu
ment."
     LimitString
#^^^^限制符的缩进. 出错! 这个脚本将不会如期运行.

echo "-----------------------------------------------------------
------------"

#  这些评论在嵌入文档范围外并不能输出

echo "Outside the here document."

exit 0

echo "This line had better not echo."  # 紧跟着个 'exit' 命令.
```

- 有些人非常聪明的使用了一个单引号(!)做为限制符. 但这并不是个好主意

```
# 这个可以运行.
cat <<!
Hello!
! Three more exclamations !!!
!


# 但是 . . .
cat <<!
Hello!
Single exclamation point follows!
!
!
# Crashes with an error message.


# 然而, 下面这样也能运行.
cat <<EOF
Hello!
Single exclamation point follows!
!
EOF
# 使用多字符限制符更为安全.
```

为嵌入文档设置这些任务有些复杂，可以考虑使用 `expect` ，一种专门用来和程序进行交互的脚本语言。

Notes: 除此之外，Dennis Benzinger 指出，使用 <<- 抑制 tab.

20 I/O 重定向

目录

- 20.1 使用 exec
- 20.2 重定向代码块
- 20.3 应用程序

有三个默认打开的文件[1], `stdin` (标准输入，键盘), `stdout` (标准输出， 屏幕) 和 `stderr` (标准错误，屏幕上输出的错误信息)。这些和任何其他打开的文件都可以被重定向。重定向仅仅意味着捕获输出文件，命令，脚本，甚至是一个脚本的代码块(样例 3-1)和(样例 3-2) 作为另一个文件，命令，程序或脚本的输入。

每个打开的文件都有特定的文件描述符。[2],而 `stdin` ， `stdout` ， `stderr` 的文件描述符分别为 0,1,2。当然了，还有附件的文件描述符 3 - 9。有时候为 `stdin` ， `stdout` ， `stderr` 临时性的复制链接分配这些附加的文件描述符会非常有用.[3]。这简化了复杂重定向和重组后的恢复(见样例 20-1)

```
COMMAND_OUTPUT >
   # 重定向标准输出到一个文件.
   # 如果文件不存在则创建，否则覆盖.

   ls -lR > dir-tree.list
   # 创建了一个包含目录树列表的文件.

: > filename
   # ">" 清空了文件.
   # 如果文件不存在，则创建了个空文件 (效果类似 'touch').
   # ":" 是个虚拟占位符, 不会有输出.

> filename
   # ">" 清空了文件.
   # 如果文件不存在，则创建了个空文件 (效果类似 'touch').
   # (结果和上述的 ": >" 一样， 但在某些 shell 环境中不能正常运行.)

COMMAND_OUTPUT >>
   # 重定向标准输出到一个文件.
   # 如果文件不存在则创建，否则新内容在文件末尾追加.
```

```
      # 单行重定向命令 (只作用于本身所在的那行):
      # --------------------------------------------------------
------------

   1>filename
      # 以覆盖的方式将 标准错误 重定向到文件 "filename."
   1>>filename
      # 以追加的方式将 标准输出 重定向到文件 "filename."
   2>filename
      # 以覆盖的方式将 标准错误 重定向到文件 "filename."
   2>>filename
      # 以追加的方式将 标准错误 重定向到文件 "filename."
   &>filename
      # 以覆盖的方式将 标准错误 和 标准输出 同时重定向到文件 "filename."
      # 在 bash 4 中才有这个新功能.

   M>N
     # "M" 是个文件描述符, 如果不明确指定，默认为 1.
     # "N" 是个文件名.
     # 文件描述符 "M" 重定向到文件 "N."
   M>&N
     # "M" 是个文件描述符, 如果不设置默认为 1.
     # "N" 是另一个文件描述符.

      #========================================================
=====================

      # 重定向 标准输出，一次一行.
      LOGFILE=script.log

      echo "This statement is sent to the log file, \"$LOGFILE\"
." 1>$LOGFILE
      echo "This statement is appended to \"$LOGFILE\"." 1>>$LOG
FILE
      echo "This statement is also appended to \"$LOGFILE\"." 1>
>$LOGFILE
      echo "This statement is echoed to stdout, and will not app
ear in \"$LOGFILE\"."
```

```
        # 这些重定向命令在每行结束后自动"重置".




        # 重定向 标准错误，一次一行.
        ERRORFILE=script.errors

        bad_command1 2>$ERRORFILE       #  Error message sent to $
ERRORFILE.
        bad_command2 2>>$ERRORFILE      #  Error message appended
to $ERRORFILE.
        bad_command3                    #  Error message echoed to
 stderr,
                                        #+ and does not appear in
$ERRORFILE.
        # 这些重定向命令每行结束后会自动“重置”.
    #==========================================================
============
```

```
    2>&1
        # 重定向 标准错误 到 标准输出.
        # 错误信息发送到标准输出相同的位置.
          >>filename 2>&1
              bad_command >>filename 2>&1
              # 同时将 标准输出 和 标准错误 追加到文件 "filename" 中 ...
          2>&1 | [command(s)]
              bad_command 2>&1 | awk '{print $5}'   # found
              # 通过管道传递 标准错误.
              # bash 4 中可以将 "2>&1 |" 缩写为 "|&".

    i>&j
        # 重定向文件描述符 i 到 j.
        # 文件描述符 i 指向的文件输出将会重定向到文件描述符 j 指向的文件

    >&j
        # 默认的标准输出 (stdout) 重定向到 j.
        # 所有的标准输出将会重定向到 j 指向的文件.
```

```
0< FILENAME
 < FILENAME
   # 从文件接收输入.
   # 类似功能命令是 ">", 经常会组合使用.
   #
   # grep search-word <filename


[j]<>filename
   #  打开并读写文件 "filename" ,
   #+ 并且分配文件描述符 "j".
   #  如果 "filename" 不存在则创建.
   #  如果文件描述符 "j" 未指定, 默认分配文件描述符 0, 标准输入.
   #
   #  这是一个写指定文件位置的应用程序.
   echo 1234567890 > File    # 写字符串到 "File".
   exec 3<> File             # 打开并分配文件描述符 3 给 "File" .
   read -n 4 <&3             # 读取 4 字符.
   echo -n . >&3             # 写一个小数点.
   exec 3>&-                 # 关闭文件描述符 3.
   cat File                  # ==> 1234.67890
   #  随机访问.



|
   # 管道.
   # 一般是命令和进程的链接工具.
   # 类似 ">", 但更一般.
   # 在连接命令，脚本，文件和程序方面非常有用.
   cat *.txt | sort | uniq > result-file
   # 所有 .txt 文件输出进行排序并且删除复制行,
   # 最终保存结果到 "result-file".
```

可以用单个命令行表示输入和输出的多个重定向或管道.

```
command < input-file > output-file
# 或者等价:
< input-file command > output-file   # 尽管这不标准.

command1 | command2 | command3 > output-file
```

更多详情见样例 16-31 and 样例 A-14.

多个输出流可以重定向到一个文件.

```
ls -yz >> command.log 2>&1
#  捕获不合法选项 "yz" 的结果到文件 "command.log."
#  因为 标准错误输出 被重定向到了文件,
#+ 任何错误信息都会在这.

#  注意, 然而, 接下来的这个案例并 "不能" 同样的结果.
ls -yz 2>&1 >> command.log
#  输出一条错误信息，但是不会写入到文件.
#  恰恰的, 命令输出(这个例子里为空)写入到文件, 但错误信息只会在 标准输出 输
出.

#  如果同时重定向 标准输出 和 标准错误输出,
#+ 命令的顺序不同会导致不同.
```

关闭文件描述符

```
n<&-
    关闭输入文件描述符 n.

0<&-, <&-
    关闭标准输入.

n>&-
    关闭输出文件描述符 n.

1>&-, >&-
    关闭标准输出.
```

子进程能继承文件描述符.这就是管道符能工作的原因.通过关闭文件描述符来防止继承 .

```
# 只重定向到 标准错误 到管道.

exec 3>&1                              # 保存当前 标准输出 "值".

ls -l 2>&1 >&3 3>&- | grep bad 3>&-    # 关闭 'grep' 文件描述符 3 (但不是 'ls').
#              ^^^^   ^^^^
exec 3>&-                              # 现在关闭它.

# 感谢, S.C.
```

更多关于 I/O 重定向详情见 Appendix F.

注意

[1] 在 UNIX 和 Linux 中, 数据流和周边外设(device files) 都被看做文件.

[2] `文件描述符` 仅仅是操作系统分配的一个可追踪的打开的文件号. 可以认为是一个简化的文件指针. 类似于 C 语言的 `文件句柄` .

[3] 当 bash 创建一个子进程的时候使用 `文件描述符 5` 会有问题. 例如 exec, 子进程继承了文件描述符 5 (详情见 Chet Ramey's 归档的 e-mail, SUBJECT: RE: File descriptor 5 is held open). 最好将这个文件描述符单独规避.

20.1 使用 exec

一个 `exec < filename` 命令重定向了 `标准输入` 到一个文件。自此所有 `标准输入` 都来自该文件而不是默认来源(通常是键盘输入)。在使用 sed 和 awk 时候这种方式可以逐行读文件并逐行解析。

样例 20-1. 使用 `exec` 重定向 标准输入

```
#!/bin/bash
# 使用 'exec' 重定向 标准输入 .


exec 6<&0          # 链接文件描述符 #6 到标准输入.
                   # .

exec < data-file   # 标准输入被文件 "data-file" 替换

read a1            # 读取文件 "data-file" 首行.
read a2            # 读取文件 "data-file" 第二行

echo
echo "Following lines read from file."
echo "-------------------------------"
echo $a1
echo $a2

echo; echo; echo

exec 0<&6 6<&-
#  现在在之前保存的位置将从文件描述符 #6 将 标准输出 恢复.
#+ 且关闭文件描述符 #6 ( 6<&- ) 让其他程序正常使用.
#
# <&6 6<&-    also works.

echo -n "Enter data  "
read b1  # 现在按预期的，从正常的标准输入 "read".
echo "Input read from stdin."
echo "----------------------"
echo "b1 = $b1"

echo

exit 0
```

同理, `exec >filename` 重定向 标准输出 到指定文件. 他将所有的命令输出通常是 标准输出 重定向到指定的位置.

`exec N > filename` 影响整个脚本或当前 shell。PID 从重定向脚本或 shell 的那时候已经发生了改变. 然而 `N > filename` 影响的就是新派生的进程，而不是整个脚本或 shell。

样例 20-2. 使用 exec 重定向标准输出

```
#!/bin/bash
# reassign-stdout.sh

LOGFILE=logfile.txt

exec 6>&1           # 链接文件描述符 #6 到标准输出.
                    # 保存标准输出.

exec > $LOGFILE     # 标准输出被文件 "logfile.txt" 替换.

# ----------------------------------------------------------- #
# 所有在这个块里的命令的输出都会发送到文件 $LOGFILE.

echo -n "Logfile: "
date
echo "-------------------------------------"
echo

echo "Output of \"ls -al\" command"
echo
ls -al
echo; echo
echo "Output of \"df\" command"
echo
df

# ----------------------------------------------------------- #

exec 1>&6 6>&-      # 关闭文件描述符 #6 恢复 标准输出.

echo
echo "== stdout now restored to default == "
echo
ls -al
echo

exit 0
```

样例 20-3. 用 exec 在一个脚本里同时重定向 标准输入 和 标准输出

```
#!/bin/bash
# upperconv.sh
# 转化指定的输入文件成大写.

E_FILE_ACCESS=70
E_WRONG_ARGS=71

if [ ! -r "$1" ]     # 指定的输入文件是否可读?
then
  echo "Can't read from input file!"
  echo "Usage: $0 input-file output-file"
  exit $E_FILE_ACCESS
fi                   #  同样的错误退出
                     #+ 等同如果输入文件 ($1) 未指定 (为什么?).

if [ -z "$2" ]
then
  echo "Need to specify output file."
  echo "Usage: $0 input-file output-file"
  exit $E_WRONG_ARGS
fi



exec 4<&0
exec < $1            # 将从输入文件读取.

exec 7>&1
exec > $2            # 将写入输出文件.
                     # 假定输出文件可写 (增加检测?).

# -----------------------------------------------
    cat - | tr a-z A-Z   # 转化大写.
#   ^^^^^                # 读取标准输入.
#           ^^^^^^^^^^   # 写到标准输出.
# 然而标准输入和标准输出都会被重定向.
# 注意 'cat' 可能会被遗漏.
# -----------------------------------------------

exec 1>&7 7>&-       # 恢复标准输出.
```

```
exec 0<&4 4<&-       # 恢复标准输入.

# 恢复后, 下面这行会预期从标准输出打印.
echo "File \"$1\" written to \"$2\" as uppercase conversion."

exit 0
```

I/O 重定向是种明智的规避 inaccessible variables within a subshell 问题的方法.

样例 20-4. 规避子 shell

```
#!/bin/bash
# avoid-subshell.sh
# Matthew Walker 的建议.

Lines=0

echo

cat myfile.txt | while read line;
                 do {
                   echo $line
                   (( Lines++ ));  #  递增变量的值趋近外层循环
                                   #  使用子 shell 会有问题.
                 }
                 done

echo "Number of lines read = $Lines"     # 0
                                         # 报错!

echo "------------------------"


exec 3<> myfile.txt
while read line <&3
do {
  echo "$line"
  (( Lines++ ));                   #  递增变量的值趋近外层循环.
                                   #  没有子 shell，就不会有问题.
}
```

```
done
exec 3>&-

echo "Number of lines read = $Lines"     # 8

echo

exit 0

# 下面的行并不在脚本里.

$ cat myfile.txt

Line 1.
Line 2.
Line 3.
Line 4.
Line 5.
Line 6.
Line 7.
Line 8.
```

20.2 重定向代码块

有如 while, until, 和 for 循环, 甚至 if/then 也可以重定向 标准输入 测试代码块. 甚至连一个函数都可以用这个方法进行重定向 (见 样例 24-11). 代码块的末尾部分的 "<" 就是用来完成这个的.

样例 20-5. while 循环的重定向

```
#!/bin/bash
# redir2.sh

if [ -z "$1" ]
then
  Filename=names.data       # 如果不指定文件名的默认值.
else
  Filename=$1
fi
#+ Filename=${1:-names.data}
#  can replace the above test (parameter substitution).

count=0

echo

while [ "$name" != Smith ]  # 为什么变量 "$name" 加引号?
do
  read name                 # 从 $Filename 读取值, 而不是 标准输入.
  echo $name
  let "count += 1"
done <"$Filename"           # 重定向标准输入到文件 $Filename.
#    ^^^^^^^^^^^^

echo; echo "$count names read"; echo

exit 0

#  注意在一些老的脚本语言中,
#+ 循环的重定向会跑在子 shell 的环境中.
```

```
#  因此, $count 返回 0, 在循环外已经初始化过值.
#  Bash 和 ksh *只要可能* 会避免启动子 shell ,
#+ 所以这个脚本作为样例运行成功.
#  (感谢 Heiner Steven 指出这点.)

#  然而 . . .
#  Bash 有时候 *能* 在 "只读的 while" 循环启动子进程 ,
#+ 不同于 "while" 循环的重定向.

abc=hi
echo -e "1\n2\n3" | while read l
     do abc="$l"
        echo $abc
     done
echo $abc

#  感谢, Bruno de Oliveira Schneider 上面的演示代码.
#  也感谢 Brian Onn 纠正了注释的错误.
```

样例 20-6. 另一种形式的 while 循环重定向

```
#!/bin/bash

# 这是之前的另一种形式的脚本.

#  Heiner Steven 提议在重定向循环时候运行在子 shell 可以作为一个变通方案
#+ 因此直到循环终止时循环内部的变量不需要保证他们的值



if [ -z "$1" ]
then
  Filename=names.data     # 如果不指定文件名的默认值.
else
  Filename=$1
fi



exec 3<&0                 # 保存标准输入到文件描述符 3.
exec 0<"$Filename"        # 重定向标准输入.
```

```
count=0
echo


while [ "$name" != Smith ]
do
  read name                  # 从重定向的标准输入($Filename)读取值.
  echo $name
  let "count += 1"
done                         #  从 $Filename 循环读
                             #+ 因为第 20 行.

#  这个脚本的早期版本在 "while" 循环 done <"$Filename" 终止
#  练习:
#  为什么这个没必要?


exec 0<&3                    # 恢复早前的标准输入.
exec 3<&-                    # 关闭临时的文件描述符 3.

echo; echo "$count names read"; echo

exit 0
```

样例 20-7. until 循环的重定向

```
#!/bin/bash
# 同先前的脚本一样, 不过用的是 "until" 循环.

if [ -z "$1" ]
then
  Filename=names.data         # 如果不指定文件的默认值.
else
  Filename=$1
fi

# while [ "$name" != Smith ]
until [ "$name" = Smith ]     # 变  !=  为 =.
do
  read name                   # 从 $Filename 读取值, 而不是标准输入.
  echo $name
done <"$Filename"             # 重定向标准输入到文件 "$Filename".
#    ^^^^^^^^^^^^

# 和之前的 "while" 循环样例相同的结果.

exit 0
```

样例 20-8. for 循环的重定向

```
#!/bin/bash

if [ -z "$1" ]
then
  Filename=names.data          # 如果不指定文件的默认值.
else
  Filename=$1
fi

line_count=`wc $Filename | awk '{ print $1 }'`
#            目标文件的行数.
#
#  非常作和不完善, 然而这只是证明 "for" 循环中的重定向标准输入是可行的
#+ 如果你足够聪明的话.
#
# 简介的做法是     line_count=$(wc -l < "$Filename")


for name in `seq $line_count`  # 回忆下 "seq" 可以输入数组序列.
# while [ "$name" != Smith ]   --   比 "while" 循环更复杂的循环   -
-
do
  read name                    # 从 $Filename 读取值, 而不是标准输入
.
  echo $name
  if [ "$name" = Smith ]       # 这需要所有这些额外的设置.
  then
    break
  fi
done <"$Filename"              # 重定向标准输入到文件 "$Filename".
#    ^^^^^^^^^^^^

exit 0
```

我们可以修改先前的样例也可以重定向循环的输出.

样例 20-9. for 循环的重定向 (同时重定向标准输入和标准输出)

```
#!/bin/bash

if [ -z "$1" ]
then
  Filename=names.data          # 如果不指定文件的默认值.
else
  Filename=$1
fi

Savefile=$Filename.new         # 报错的结果的文件名.
FinalName=Jonah                # 停止 "read" 的终止字符.

line_count=`wc $Filename | awk '{ print $1 }'`  # 目标文件行数.



for name in `seq $line_count`
do
  read name
  echo "$name"
  if [ "$name" = "$FinalName" ]
  then
    break
  fi
done < "$Filename" > "$Savefile"     # 重定向标准输入到文件 $Filena
me,
#    ^^^^^^^^^^^^^^^^^^^^^^^^^^^        并且报错结果到备份文件.

exit 0
```

样例 20-10. if/then test的重定向

```
#!/bin/bash

if [ -z "$1" ]
then
  Filename=names.data   # 如果不指定文件的默认值.
else
  Filename=$1
fi

TRUE=1

if [ "$TRUE" ]          # if true    和   if :   都可以工作.
then
 read name
 echo $name
fi <"$Filename"
#  ^^^^^^^^^^^^

# 只读取文件的首行.
# "if/then" test 除非嵌入在循环内部否则没办法迭代.

exit 0
```

样例 20-11. 上述样例的数据文件 names.data

```
Aristotle
Arrhenius
Belisarius
Capablanca
Dickens
Euler
Goethe
Hegel
Jonah
Laplace
Maroczy
Purcell
Schmidt
Schopenhauer
Semmelweiss
Smith
Steinmetz
Tukhashevsky
Turing
Venn
Warshawski
Znosko-Borowski

#+ 这是 "redir2.sh", "redir3.sh", "redir4.sh", "redir4a.sh", "redir5.sh" 的数据文件.
```

代码块的标准输出的重定向影响了保存到文件的输出. 见样例 样例 3-2.

嵌入文档 是种特别的重定向代码块的方法. 既然如此,它使得在 while 循环的标准输入里传入嵌入文档的输出变得可能.

```
# 这个样例来自 Albert Siersema
# 得到了使用许可 (感谢!).

function doesOutput()
 # 当然这也是个外部命令.
 # 这里用函数进行演示会更好一点.
{
  ls -al *.jpg | awk '{print $5,$9}'
}


nr=0          #  我们希望在 'while' 循环里可以操作这些
totalSize=0   #+ 并且在 'while' 循环结束时看到改变.

while read fileSize fileName ; do
  echo "$fileName is $fileSize bytes"
  let nr++
  totalSize=$((totalSize+fileSize))   # Or: "let totalSize+=file
Size"
done<<EOF
$(doesOutput)
EOF

echo "$nr files totaling $totalSize bytes"
```

20.3 应用程序

使用 I/O 重定向可以同时解析和固定命令输出的片段(see 样例 15-7). 这也使得可以生成报告和日志文件.

样例 20-12. 日志记录事件

```
#!/bin/bash
# logevents.sh
# 作者: Stephane Chazelas.
# 用于 ABS 许可指南.

# 事件记录到文件.
# 必须 root 身份执行 (可以写入 /var/log).

ROOT_UID=0     # 只有 $UID 为 0 的用户具有 root 权限.
E_NOTROOT=67   # 非 root 会报错.


if [ "$UID" -ne "$ROOT_UID" ]
then
  echo "Must be root to run this script."
  exit $E_NOTROOT
fi


FD_DEBUG1=3
FD_DEBUG2=4
FD_DEBUG3=5

# === 取消下面两行注释来激活脚本. ===
# LOG_EVENTS=1
# LOG_VARS=1


log()  # 时间和日期写入日志文件.
{
echo "$(date)  $*" >&7     # *追加* 日期到文件.
```

```
#     ^^^^^^^  命令替换
                           # 见下文.
}




case $LOG_LEVEL in
 1) exec 3>&2         4> /dev/null 5> /dev/null;;
 2) exec 3>&2         4>&2         5> /dev/null;;
 3) exec 3>&2         4>&2         5>&2;;
 *) exec 3> /dev/null 4> /dev/null 5> /dev/null;;
esac

FD_LOGVARS=6
if [[ $LOG_VARS ]]
then exec 6>> /var/log/vars.log
else exec 6> /dev/null                     # 清空输出.
fi

FD_LOGEVENTS=7
if [[ $LOG_EVENTS ]]
then
  # exec 7 >(exec gawk '{print strftime(), $0}' >> /var/log/even
t.log)
  # 上述行在最近高于 bash 2.04 版本会失败，为什么?
  exec 7>> /var/log/event.log              # 追加到 "event.log".
  log                                      # 写入时间和日期.
else exec 7> /dev/null                     # 清空输出.
fi

echo "DEBUG3: beginning" >&${FD_DEBUG3}

ls -l >&5 2>&4                             # 命令1 >&5 2>&4

echo "Done"                                # 命令2

echo "sending mail" >&${FD_LOGEVENTS}
# 输出信息 "sending mail" 到文件描述符 #7.
```

```
exit 0
```

第二十二章. 限制模式的Shell

限制模式下被禁用的命令

- 在限制模式下运行一个脚本或部分脚本将禁用一些命令，尽管这些命令在正常模式下是可用的。这是个安全措施，可以限制脚本用户的权限，减少运行脚本可能带来的损害。

被禁用的命令和功能：

- 使用 *cd* 来改变工作目录。
- 修改 *$PATH*, *$SHELL*, *$BASH_ENV* 或 *$ENV* 等环境变量
- 读取或修改 *$SHELLOPTS*，shell环境选项。
- 输出重定向。
- 调用包含 / 的命令。
- 调用 exec 来替代shell进程。
- 其他各种会造成混乱或颠覆脚本用途的命令。
- 在脚本中跳出限制模式。

例 **22-1.** 在限制模式运行脚本

```
#!/bin/bash

#  在脚本开头用"#!/bin/bash -r"
#+ 可以让整个脚本在限制模式运行。

echo

echo "改变目录。"
cd /usr/local
echo "现在是在 `pwd`"
echo "回到家目录。"
cd
echo "现在是在 `pwd`"
echo

# 到此为止一切都是正常的，非限制模式。
```

```
set -r
# set --restricted 效果相同。
echo "==> 现在是限制模式 <=="

echo
echo

echo "在限制模式试图改变目录。"
cd ..
echo "依旧在 `pwd`"

echo
echo

echo "\$SHELL = $SHELL"
echo "试图在限制模式改变Shell 。"
SHELL="/bin/ash"
echo
echo "\$SHELL= $SHELL"

echo
echo

echo "试图在限制模式重定向输出内容。"
ls -l /usr/bin > bin.files
ls -l bin.files    # 尝试列出试图创建的文件。

echo

exit 0
```

第二十三章. 进程替换

用管道 将一个命令的 `标准输出` 输送到另一个命令的 `标准输入` 是个强大的技术。但是如果你需要用管道输送多个命令的 `标准输出` 怎么办？这时候 进程替换 就派上用场了。

进程替换 把一个（或多个）进程 的输出送到另一个进程的 `标准输入` 。

样板 命令列表要用括号括起来

```
>(command_list)
<(command_list)
```

进程替换使用 `/dev/fd/<n>` 文件发送括号内进程的结果到另一个进程。[1]

"<"或">"与括号之间没有空格，加上空格或报错。

```
bash$ echo >(true)
/dev/fd/63

bash$ echo <(true)
/dev/fd/63

bash$ echo >(true) <(true)
/dev/fd/63 /dev/fd/62

bash$ wc <(cat /usr/share/dict/linux.words)
 483523  483523 4992010 /dev/fd/63

bash$ grep script /usr/share/dict/linux.words | wc
    262     262    3601

bash$ wc <(grep script /usr/share/dict/linux.words)
    262     262    3601 /dev/fd/63
```

Bash用两个文件描述符创建管道， `--fIn 和 fOut--` 。true 的 `标准输入` 连接 fOut(dup2(fOut, 0))，然后Bash 传递一个 `/dev/fd/fIn` 参数给 **echo** 。在不使用 `/dev/fd/<n>` 的系统里，Bash可以用临时文件（感谢 S.C. 指出这点）。

进程替换可以比较两个不同命令的输出，或者同一个命令使用不同选项的输出。

```
bash$ comm <(ls -l) <(ls -al)
total 12
-rw-rw-r--    1 bozo bozo       78 Mar 10 12:58 File0
-rw-rw-r--    1 bozo bozo       42 Mar 10 12:58 File2
-rw-rw-r--    1 bozo bozo      103 Mar 10 12:58 t2.sh
        total 20
        drwxrwxrwx    2 bozo bozo     4096 Mar 10 18:10 .
        drwx------   72 bozo bozo     4096 Mar 10 17:58 ..
        -rw-rw-r--    1 bozo bozo       78 Mar 10 12:58 File0
        -rw-rw-r--    1 bozo bozo       42 Mar 10 12:58 File2
        -rw-rw-r--    1 bozo bozo      103 Mar 10 12:58 t2.sh
```

进程替换可以比较两个目录的内容——来检查哪些文件在这个目录而不在那个目录。

```
diff <(ls $first_directory) <(ls $second_directory)
```

进程替换的一些其他用法：

```
read -a list < <( od -Ad -w24 -t u2 /dev/urandom )
#  从 /dev/urandom 读取一个随机数列表
#+ 用 "od" 处理
#+ 输送到 "read" 的标准输入. . .
#  来自 "insertion-sort.bash" 示例脚本。
#  致谢：JuanJo Ciarlante。
```

```
PORT=6881   # bittorrent（BT端口）

#  扫描端口，确保没有恶意行为
netcat -l $PORT | tee>(md5sum ->mydata-orig.md5) |
gzip | tee>(md5sum - | sed 's/-$/mydata.lz2/'>mydata-gz.md5)>myd
ata.gz

#  检查解压缩结果：
  gzip -d<mydata.gz | md5sum -c mydata-orig.md5)
#  对原件的MD5校验用来检查标准输入，并且探测压缩当中出现的问题。

#  Bill Davidsen 贡献了这个例子
#+ （ABS指南作者做了轻微修改）。
```

```
cat <(ls -l)
# 等价于    ls -l | cat

sort -k 9 <(ls -l /bin) <(ls -l /usr/bin) <(ls -l /usr/X11R6/bin
)
#  列出 3 个主要 'bin' 目录的文件，按照文件名排序。
#  注意，有三个（数一下）单独的命令输送给了 'sort'。

diff <(command1) <(command2)    # 比较命令输出结果的不同之处。

tar cf >(bzip2 -c > file.tar.bz2) $directory_name

#  调用 "tar cf /dev/fd/?? $directory_name"，然后 "bzip2 -c > file
.tar.bz2"。
#
#  因为 /dev/fd/<n> 系统特性
#  不需要在两个命令之间使用管道符
#
#  这个可以模拟
#
bzip2 -c < pipe > file.tar.bz2&
tar cf pipe $directory_name
rm pipe
#   或者
exec 3>&1
tar cf /dev/fd/4 $directory_name 4>&1 >&3 3>&- | bzip2 -c > file
.tar.bz2 3>&-
exec 3>&-

# 致谢：Stéphane Chazelas
```

在子shell中 echo 命令用管道输送给 while-read 循环时会出现问题，下面是避免的方法：

例23-1 不用 **fork** 的代码块重定向。

```
#!/bin/bash

#  wr-ps.bash: 使用进程替换的 while-read 循环。
```

```
#  示例由 Tomas Pospisek 贡献。
# （ABS指南作者做了大量改动。）

echo

echo "random input" | while read i
do
  global=3D": Not available outside the loop."
  # ... 因为在子 shell 中运行。
done

echo "\$global (从子进程之外) = $global"
# $global (从子进程之外) =

echo; echo "--"; echo

while read i
do
  echo $i
  global=3D": Available outside the loop."
  # ... 因为没有在子 shell 中运行。
done < <( echo "random input" )
#    ^ ^

echo "\$global (使用进程替换) = $global"
#  随机输入
#  $global (使用进程替换)= 3D: Available outside the loop.


echo; echo "##########"; echo



# 同样道理 . . .

declare -a inloop
index=0
cat $0 | while read line
do
```

```
  inloop[$index]="$line"
  ((index++))
  # 在子 shell 中运行，所以 ...
done
echo "OUTPUT = "
echo ${inloop[*]}           # ... 什么也没有显示。


echo; echo "--"; echo


declare -a outloop
index=0
while read line
do
  outloop[$index]="$line"
  ((index++))
  # 没有在子 shell 中运行，所以 ...
done < <( cat $0 )
echo "OUTPUT = "
echo ${outloop[*]}          # ... 整个脚本的结果显示出来。

exit $?
```

下面是个类似的例子。

例 **23-2.** 重定向进程替换的输出到一个循环内

```
#!/bin/bash
# psub.bash
#  受 Diego Molina 启发（感谢！）。

declare -a array0
while read
do
  array0[${#array0[@]}]="$REPLY"
done < <( sed -e 's/bash/CRASH-BANG!/' $0 | grep bin | awk '{pri
nt $1}' )
#  由进程替换来设置'read'默认变量（$REPLY）。
#+ 然后将变量复制到一个数组。

echo "${array0[@]}"

exit $?

# ====================================== #
# 运行结果：
bash psub.bash

#!/bin/CRASH-BANG! done #!/bin/CRASH-BANG!
```

一个读者发来一个有趣的进程替换例子，如下：

```
# SuSE 发行版中提取的脚本片段：

# --------------------------------------------------------------
#
while read  des what mask iface; do
# 一些命令 ...
done < <(route -n)
#    ^ ^  第一个 < 是重定向，第二个是进程替换。

#  为了测试，我们让它来做点儿事情。
while read  des what mask iface; do
  echo $des $what $mask $iface
done < <(route -n)
```

```
# 输出内容:
# Kernel IP routing table
# Destination Gateway Genmask Flags Metric Ref Use Iface
# 127.0.0.0 0.0.0.0 255.0.0.0 U 0 0 0 lo
# -------------------------------------------------------------
#

#  正如 Stéphane Chazelas 指出的,
#+ 一个更容易理解的等价代码如下:
route -n |
  while read des what mask iface; do   # 通过管道输出设置的变量
    echo $des $what $mask $iface
  done  #  这段代码的结果更上面的相同。
        #  但是,Ulrich Gayer 指出 . . .
        #+ 这段简化版等价代码在 while 循环里用了子 shell,
        #+ 因此当管道终止时变量都消失了。

# -------------------------------------------------------------
#

#  然而,Filip Moritz 说上面的两个例子有一个微妙的区别,
#+ 见下面的代码

(
route -n | while read x; do ((y++)); done
echo $y # $y is still unset

while read x; do ((y++)); done < <(route -n)
echo $y # $y has the number of lines of output of route -n
)

#  更通俗地说(译者注:原文本行少了注释符)
(
: | x=x
# 似乎启动了子 shell ,就像
: | ( x=x )
# 而
x=x < <(:)
# 并没有。
)
```

```
#  这个方法在解析 csv 和类似格式时很有用。
#  也就是在效果上，原始 SuSE 系统的代码片段就是做这个用的。
```

注解 [1] 这个与命名管道（使用临时文件）的效果相同，而且事实上，进程替换也曾经用过命名管道。

第二十六章. 列表结构

and 列表 和 *or* 列表 结构提供了连续执行若干命令的方法，可以有效地替换复杂的嵌套 if/then ，甚至 case 语句。

链接多个命令

and 列表

```
command-1 && command-2 && command-3 && ... command-n
```

只要前一个命令返回 *true*（即 0），每一个命令就依次执行。当第一个 *false*（即非0）返回时，命令链条即终止（第一个返回 *false* 的命令是最后一个执行的）。

在YongYe早期版本的俄罗斯方块游戏脚本里，一个有趣的双条件 *and* 列表 用法：

```
equation()

{   # core algorithm used for doubling and halving the coordinate
s
    [[ ${cdx} ]] && ((y=cy+(ccy-cdy)${2}2))
    eval ${1}+=\"${x} ${y} \"
}
```

例 **26-1.** 使用 ***and*** 列表 来测试命令行参数

```
#!/bin/bash
# and list

if [ ! -z "$1" ] && echo "Argument #1 = $1" && [ ! -z "$2" ] &&
\
#                ^^                           ^^               ^^
echo "Argument #2 = $2"
then
  echo "At least 2 arguments passed to script."
  # 链条内的所有命令都返回 true。
else
  echo "Fewer than 2 arguments passed to script."
  # 链条内至少有一个命令返回 false。
fi
# 注意： "if [ ! -z $1 ]" 是好用的，但是宣传与之等同的
#   "if [ -n $1 ]" 并不好用。
#  不过，用引号就能解决问题，
#   "if [ -n "$1" ]" 好用（译者注：原文本行内第一个引号位置错了）。
#           ^  ^    小心!
# 被测试的变量放在引号内总是最好的选择。



# 下面的代码功能一样，用的是“纯粹”的 if/then 语句。
if [ ! -z "$1" ]
then
  echo "Argument #1 = $1"
fi
if [ ! -z "$2" ]
then
  echo "Argument #2 = $2"
  echo "At least 2 arguments passed to script."
else
  echo "Fewer than 2 arguments passed to script."
fi
# 比起用“and 列表”要更长、更笨重。



exit $?
```

例 **26-2.** 使用 ***and*** 列表 来测试命令行参数**2**

```
#!/bin/bash

ARGS=1          # 预期的参数数量。
E_BADARGS=85    # 参数数量错误时返回的值。

test $# -ne $ARGS && \
#    ^^^^^^^^^^^^ 条件 #1
echo "Usage: `basename $0` $ARGS argument(s)" && exit $E_BADARGS
#                                             ^^
#  如果条件 #1 结果为 true (传递给脚本的参数数量错误),
#+ 那么执行本行剩余的命令，脚本终止。

# 下面的代码行只有在上面的测试失败时才执行。
echo "Correct number of arguments passed to this script."

exit 0

#  如果要检查退出值，脚本终止后运行 "echo $?"。
```

当然，*and* 列表 也可以给变量设置默认值。

```
arg1=$@ && [ -z "$arg1" ] && arg1=DEFAULT

              # 如果有命令行参数，则把参数值赋给 $arg1 。
              # 但是... 如果没有参数，则使用DEFAULT给 $arg1 赋值。
```

or 列表

```
command-1 || command-2 || command-3 || ... command-n
```

只要前一个命令返回*false*，每一个命令就依次执行。当第一个*true* 返回时，命令链条即终止（第一个返回*true* 的命令是最后一个执行的）。很明显它与“and 列表”相反。

例 26-3. *or* 列表 与 *and* 列表 结合使用

```
#!/bin/bash

#  delete.sh, 不那么巧妙的文件删除工具。
#  用法： delete 文件名

E_BADARGS=85

if [ -z "$1" ]
then
  echo "Usage: `basename $0` filename"
  exit $E_BADARGS  # No arg? Bail out.
else
  file=$1          # Set filename.
fi



[ ! -f "$file" ] && echo "File \"$file\" not found. \
Cowardly refusing to delete a nonexistent file."
# AND 列表，如果文件不存在则显示出错信息。
# 注意，echo 消息内容分成了两行，中间通过转义符（\）连接。

[ ! -f "$file" ] || (rm -f $file; echo "File \"$file\" deleted."
)
# OR 列表，删除存在的文件。

# 注意上面的逻辑颠倒。 Note logic inversion above.
# “AND 列表” 在得到 true 时执行, “OR 列表”在得到 false 时执行。

exit $?
```

 如果 *or* 列表 第一个命令返回 true，它会执行。

```
# ==> 下面的代码段来自 /etc/rc.d/init.d/single
#+==> 作者 Miquel van Smoorenburg
#+==> 说明了 "and" 和 "or" 列表。
# ==> 带箭头的注释是本文作者添加的。

[ -x /usr/bin/clear ] && /usr/bin/clear
  # ==> 如果 /usr/bin/clear 存在, 则调用它。
  # ==> 调用命令之前检查它是否存在，
  #+==> 可以避免出错消息和其他怪异的结果。

  # ==> . . .

#  If they want to run something in single user mode, might as w
ell run it...
for i in /etc/rc1.d/S[0-9][0-9]* ; do
        # 检查脚本是否存在。
        [ -x "$i" ] || continue
  # ==> 如果对应的文件在 $PWD 里*没有*找到，
  #+==> 则跳回到循环顶端“继续运行”。

        # 丢弃备份文件和 rpm 生成的文件。
        case "$1" in
                *.rpmsave|*.rpmorig|*.rpmnew|*~|*.orig)
                        continue;;
        esac
        [ "$i" = "/etc/rc1.d/S00single" ] && continue
  # ==> 设置脚本名，但先不要执行
        $i start
done

  # ==> . . .
```

 *and* 列表 或 *or* 列表 的退出状态就是最后一个执行的命令的退出状态。

聪明地结合 *and* 列表 和 *or* 列表 是可能的，但是程序逻辑会很容易地变得令人费解，需要密切注意操作符优先规则，而且，会带来大量的调试工作。

```
false && true || echo false          # false

# 下面的代码结果相同
( false && true ) || echo false      # false
# 但这个就不同了
false && ( true || echo false )      # (什么都不显示)

#  注意语句是从左到右组合和解释的。

#  通常情况下最好避免这种复杂性。

#  感谢, S.C.
```

例 A-7 和 例 7-4 解释了用 *and* 列表 / *or* 列表 来测试变量。

25. 别名

Bash `别名` 本质上不外乎是键盘上的快捷键，缩写呢是避免输入很长的命令串的一种手段.举个例子, 在 ~/.bashrc 文件中包含别名 `lm="ls -l | more"`, 而后每个命令行输入的 lm [1] 将会自动被替换成 `ls -l | more`. 这可以节省大量的命令行输入和避免记住复杂的命令和选项. 设定别名 `rm="rm -i"` (交互的删除模式) 防止无意的删除重要文件，也许可以少些悲痛.

脚本中别名作用十分有限. 如果别名可以有一些 C 预处理器的功能会更好, 例如宏扩展, 但不幸的是 bash 别名中没有扩展参数. [2] 另外, 脚本在 "复合结构" 中并不能扩展自身的别名，例如 if/then, 循环和函数. 另一个限制是，别名不能递归扩展. 基本上是我们无论怎么喜欢用别名都不如函数 function 来的更有效.

样例 25-1. 脚本中的别名

```
#!/bin/bash
# alias.sh

shopt -s expand_aliases
# 必须设置此选项, 否则脚本不能别名扩展.



# 首先来点好玩的东西.
alias Jesse_James='echo "\"Alias Jesse James\" was a 1959 comedy
 starring Bob Hope."'
Jesse_James

echo; echo; echo;

alias ll="ls -l"
# 可以任意使用单引号 (') 或双引号 (") 把别名括起来.

echo "Trying aliased \"ll\":"
ll /usr/X11R6/bin/mk*   #* 别名可以运行.

echo
```

```
directory=/usr/X11R6/bin/
prefix=mk*  # See if wild card causes problems.
echo "Variables \"directory\" + \"prefix\" = $directory$prefix"
echo

alias lll="ls -l $directory$prefix"

echo "Trying aliased \"lll\":"
lll         # 所有 /usr/X11R6/bin 文件清单以 mk 开始.
# 别名可以处理连续的变量 -- 包含 wild card -- o.k.




TRUE=1

echo

if [ TRUE ]
then
  alias rr="ls -l"
  echo "Trying aliased \"rr\" within if/then statement:"
  rr /usr/X11R6/bin/mk*   #* 结果报错!
  # 别名在复合的表达式中并没有生效.
  echo "However, previously expanded alias still recognized:"
  ll /usr/X11R6/bin/mk*
fi

echo

count=0
while [ $count -lt 3 ]
do
  alias rrr="ls -l"
  echo "Trying aliased \"rrr\" within \"while\" loop:"
  rrr /usr/X11R6/bin/mk*   #* 这里的别名也没生效.
                           #  alias.sh: 行 57: rrr: 命令未找到
  let count+=1
done
```

```
echo; echo

alias xyz='cat $0'   # 列出了自身.
                     # 注意强引.
xyz
#  这看起来能工作,
#+ 尽管 bash 文档不介意这么做.
#
#  然而, Steve Jacobson 指出,
#+ "$0" 参数的扩展在上面的别名申明后立刻生效.

exit 0
```

取消别名的命令删除之前设置的别名.

样例 25-2. unalias: 设置和取消一个别名

```
#!/bin/bash
# unalias.sh

shopt -s expand_aliases  # 开启别名扩展.

alias llm='ls -al | more'
llm

echo

unalias llm              # 取消别名.
llm
# 'llm' 不再被识别后的报错信息.

exit 0
bash$ ./unalias.sh
total 6
drwxrwxr-x    2 bozo     bozo         3072 Feb  6 14:04 .
drwxr-xr-x   40 bozo     bozo         2048 Feb  6 14:04 ..
-rwxr-xr-x    1 bozo     bozo          199 Feb  6 14:04 unalias.
sh

./unalias.sh: llm: 命令未找到
```

注意

[1] ... 作为命令行的第一个词. 显然别名只在命令的开始有意义. [2] 然而, 别名可用来扩展位置参数.